Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Permissioned Blockchains and Distributed Databases: A Performance Study

Sara Bergman[1,2] | Mikael Asplund*[2] | Simin Nadjm-Tehrani[2]

[1]Microsoft Corporation, Norway
[2]Department of Computer and Information Science, Linköping University, Sweden

Correspondence
*Mikael Aplund, Linköping University, Dept. of Comp. and Inf. Science, SE-581 83 Linköping, Sweden Email: mikael.asplund@liu.se

Present Address
This is sample for present address text this is sample for present address text

Summary

Blockchains are becoming mainstream and new applications of blockchains are continuously being presented. Permissioned blockchains promise to remove some of the downsides of the first generation of blockchains and provide more efficient and faster operation. But can they match traditional large-scale databases? In this work we take a pure performance-oriented approach and compare two popular frameworks, Hyperledger Fabric and Apache Cassandra as representatives of permissioned blockchains and distributed databases respectively. We compare their latency for varying workloads and network sizes. The results show that for small systems, blockchains can start to compete with traditional databases, but also that the difference in consistency models and differences in setup can have a large impact on the resulting performance.

KEYWORDS:
Blockchains, Databases, Latency, Fabric, Cassandra

1 | INTRODUCTION

The benefits of building future distributed systems on top of blockchains are being explored among a much wider range of applications than the original cryptocurrency application in which they were promoted [7]. The recent propositions of adopting blockchains in diverse domains with widely varying requirements, include insurance, land registry, journalism, supply-chain management, food safety, and have made it obvious that a one-size-fits-all approach for accessing and updating information in a distributed manner where trust cannot be fully assumed does not make sense.

A major diversion from the public (permissionless) blockchains has recently been proposed [3] where the arguments for a *permissioned* blockchain are presented. These are blockchains where a mere reliance on the identity of the peers will exist, but the transactions are not trusted to be recorded by a centralised authority.

As an example, a distributed ledger, Hyperledger Fabric (from now on called simply Fabric), is proposed as an open source platform where various parties can initiate transactions and validate them in a transparent manner. Among others, a platform for implementing permissioned blockchains with pluggable components makes the adapatability of the platform plausible. The basic services that Fabric provides range over an identity provision with cryptographic membership service, an ordering service, an isolated execution environment for various contracts, and a dissemination service. Compared to classical replicated databases where transactions are ordered at each node and then subjected to local execution, Fabric has been built around an execute-order-validate architecture. The claim is that this will provide a means of combining the security, performance, and consistency requirements in such distributed applications.

On the other hand, critics of the blockchain idea claim that it is a hype with no real technical improvement over existing techniques. Blockchains have been criticized of being slow [15], potentially insecure [6], or just that in most cases it is simply not worth the trouble [16]. The alternative, for example to add a cryptographic layer for non-repudiation on an existing distributed database would perhaps meet all the requirements but with better performance.

This paper focuses on the performance aspect of the permissioned blockchains. Specifically, we address the question: what is the benefit that a permissioned blockchain brings, simply in terms of performance, if we compare with a distributed database architectures that have been subject to many years of optimisation?

The paper adopts an experimental approach to perform a controlled benchmarking on two selected platforms. This is done by building a common synthetic application that creates and accesses transaction outcomes in a distributed manner. Among several considered platforms, Cassandra and Fabric were selected for exposure to similar loads and transaction characteristics. The latency of read and write operations is studied under varying network size and load. To the best of our knowledge this is the first published comparison of permissioned blockchains and distributed databases in terms of performance.

The contribution of the paper are as follows:

- A brief comparison of five blockchain implementation frameworks and four database frameworks before selecting the two candidate platforms for experimentation.
- Studying the insert and read latencies of the two platforms for similar system sizes and workloads
- Comparing the scalability of the platforms (in a restrained environment with similar resources) as the load mix and volume changes, for up to 35 participating nodes.

Our work indicates that each of the selected frameworks have benefits in some setting. In particular, we found that since Fabric is built to run isolated contracts inside the implementation mechanism Docker it is optimized to utilize Docker in a more effective way. Because of this its overhead is smaller for Fabric.

Cassandra performs better if we ignore the container interaction part of the invocation. Therefore, if the Docker-initiated Cassandra operations can be optimised to the same extent as Fabric's, then Cassandra may provide a more effective transaction service. However, for small networks and moderate loads, the difference between the two systems is quite small.

The rest of the paper is organized as follows. Section 2 provides a short overview of some existing permissioned blockchains and distributed databases, and explains the rationale for choosing Fabric and Cassandra as representatives in this performance study. Section 3 contains a background on the Hyperledger Fabric and Cassandra frameworks which is needed to understand the experimental design and results. The experiment design is described in Section 4, followed by the results in Section 5. Section 6 describes related work and finally, the discussion and conclusion are contained in Sections 7 and 8 respectively.

2 | REVIEW OF AVAILABLE FRAMEWORKS

As a pre-study to the performance comparison presented in this paper, we provide an overview of available permissioned blockchains and distributed databases, and select one representative in each category. The purpose of this study is two-fold. First, by carefully and systematically selecting two frameworks that match the chosen criteria, we provide a stronger foundation for understanding what this comparison can say about permissioned blockchains and distributed databases in general. Second, there are many variations of deployment and potential use-case requirements which can be tuned and adjusted so that a particular framework outperforms the others. In this paper we are interested in the intersection of requirements where both types of frameworks could be used. Therefore, we select two frameworks that are as comparable as possible with regards to potential use-cases and architectural style. Since blockchains can be considered more specialized than distributed databases at large, we first select a framework from this category, and then find a distributed database that can be configured to match it.

We start the section by describing the selection criteria we have considered, followed by one subsection for blockchain frameworks and one for distributed databases.

2.1 | Criteria

To determine appropriate selection criteria we started from two basic requirements. It should be possible to deploy and configure a solution on current platforms with reasonable effort, meaning that there should be documentation, and active development of the project. Moreover, the study should be meaningful and to the largest extent possible founded on existing research. Therefore, we defined the following criteria to be used both for the blockchain and database frameworks. The criteria were evaluated during May 2018.

- *There is publicly available documentation of the framework.* This criterion was chosen to ensure that the framework could be deployed and configured.
- *Updates to the framework have been released during 2018.* This criterion was chosen to ensure that the framework is compatible with current software environments (e.g., libraries, operating system etc).
- *The performance of framework have been studied and reported in scientific literature.* This criterion was chosen to be able to validate our measurements with what has been previously reported.
- *The underlying architecture is peer-to-peer.* This criterion was chosen to limit the scope of the study to systems that emphasize a distributed deployment (in the spirit of blockchains).

In addition to these criteria we have two criteria that are specific for the blockchain frameworks:

- *The permission property is be permissioned or permissionable.* This criterion was chosen to limit the scope of the study to frameworks that can support a permissioned operation.
- *The blockchain scope is private or possible to configure to private.* This criterion was chosen to limit the scope of the study to frameworks that support private operation.

For the distributed databases we also consider which replication and consistency strategy that is used. Since most blockchains by design provide a consensus layer to ensure some kind of active consistency, it would be preferable if the database system also uses active replication.

2.2 | Permissioned Blockchain Frameworks

We selected five major open source blockchain frameworks to analyze further.

MultiChain

MultiChain is a framework for private and permissionable blockchains presented in a white paper by Greenspan [11]. 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.

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].

OpenChain

OpenChain is an open-source framework for distributed ledgers which leverages blockchain technology. It is a configurable approach for realising blockchains on top of a client-server architecture. According to its 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.

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].

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 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.

Choosing a Permissioned Blockchain Framework

Table 1 lists the chosen blockchain frameworks together with their compatibility with the desired properties.

OpenChain has the wrong architecture and HydraChain isn't an active project which makes both of them disqualified. As can be seen in Table 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 several researchers including Androulaki et al. [3], Dinh et al. [9], and Thakkar et al. [19]. The consensus process is also benchmarked by Sukhwani et al. [18]. For these reasons Hyperledger Fabric was chosen as the permissioned blockchain.

2.3 | Distributed Database Frameworks

We selected four major open source distributed database frameworks to analyze further.

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. If the primary fails, a leader election protocol will ensure that one of the secondary nodes take over as primary. MongoDB is open-source and supports over 10 programming languages.

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

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

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

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

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

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

[6] https://hadoop.apache.org

TABLE 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

manage storage. Since version 2.0 of HDFS the namespace node can be replicated using a primary-backup strategy[7].

HBase

Base 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[8]. HBase is an extension of HDFS and therefore also runs of a master-slave architecture. Replication is supported using a primary-backup approach [9].

Apache Cassandra

Apache Cassandra is a NoSQL distributed database originally developed by Facebook to accommodate its growing storage need [13]. Every node is identical in Cassandra and it is a fully distributed system running on a peer to peer network. It supports tunable consistency levels, including linearizable consistency through active replication with the help of an extension of the Paxos protocol.

Choosing a Distributed Database Framework

The investigated frameworks and their compatibility to the requirements are listed in Table 2. Both HBase and HDFS are tuned for very large datasets and better for batch processing, whereas we focus on smaller datasets with more strict requirements on consistency in presence of concurrent writes. MongoDB matches most of the given criteria, however the replication and consistency model follows a primary-backup approach as opposed to the active replication which can be configured in Cassandra. This is important since the distributed database needs to be configurable to work as similarly to Fabric as possible. Since Cassandra matches all given criteria it was chosen as the distributed database.

3 | OVERVIEW OF THE CHOSEN FRAMEWORKS

This section covers the basics about the architecture and operations of Fabric and Cassandra. This information will be needed to understand the choices made in the experiment design as well as some of the results.

3.1 | Hyperledger Fabric

The nodes which form the Fabric 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.
- Peers - Peers validate transactions from the ordering service and maintain both the state and a copy of the ledger. Peers can also take the special role of endorsement peer. The number of endorsement peers is determined by the endorsement policy, which is set by the developer.
- Orderer - All the orderer nodes collectively run the ordering service and uses a shared communication channel between clients and peers. The number of ordering nodes is small compared to the number of peers. The ordering service ensures that transactions are totally ordered on the blockchain[3]. The ordering service enforces the consensus mechanism of Fabric and can be implemented in different ways. 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, which is not intended for production and should therefore not be used when benchmarking. The second is an ordering service which uses Apache Kafka and Apache Zookeeper and is built to be used in production.

3.1.1 | 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]. 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 transaction flow is described in detail both in

[7] https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html
[8] http://hbase.apache.org/book.html7B%5C#%7Darch.overview
[9] http://hbase.apache.org/book.html#_cluster_replication

TABLE 2 Overview of frameworks for distributed databases

Name	Evaluated in literature	Architecture	Replication strategy	Documentation	Active
MongoDB	Yes	P2P	Primary-backup	Yes	Yes
HDFS	Yes	Master-slave	Primary-backup	Yes	Yes
HBase	Yes	Master-slave	Primary-backup	Yes	Yes
Cassandra	Yes	P2P	Tunable	Yes	Yes

the documentation for Fabric and by Androulaki et al. [3], the developers of Fabric.

The first phase is the execution phase, which comprises of three steps. Firstly the client sends a transaction proposal to a set of endorsement peers. 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 only return an endorsement message to the client consisting of all the state updates the transaction proposal caused. The client collects endorsements until it has enough to satisfy the endorsement policy. When the client has successfully collected enough correct endorsements, it creates a transaction which it sends to the ordering service.

This step is the start of the next phase, the ordering phase, in which the ordering service places all the incoming transactions from clients in a total order. 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`. The ordering service then broadcasts the hash chain of blocks to all peers, including the endorsement peers.

The last part of the transaction flow is the validation phase. All peers receive the hash chain of blocks from the ordering service. Each block is subject to validation on the peer, since there might be faulty transactions on the blocks. The first step is to evaluate the endorsement policy. If the endorsement policy is not fulfilled, the transaction is considered invalid. The next step of validation is to check if the version of the state changes in the endorsements compared to the peers' local state. If the versions don't match the transaction is considered invalid. All the effects of an invalid transaction are ignored but the transaction isn't removed from the block. The last step is to append the block to the peers' local ledger and update the state by writing the state changes to the peers' local state.

3.1.2 | Reading data

Reading data, or querying the ledger, is much simpler than adding a new transaction. Queries can be invoked by a client using chaincode, which is the program code which implements the application logic [3], and the chaincode communicates with the peer over a secure channel. The peer in turn queries the local state and returns the response to the chaincode. Then the chaincode executes the chaincode logic and returns the answer to the client via the peer.

3.2 | Cassandra

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 [13]. Cassandra has its own querying language, cql, which is the only way to interact with the system.

3.2.1 | Lightweight Transactions

Cassandra uses an extended version of Paxos to supports linearizable consistency for a type of operations called lightweight transactions (LWT)[10]. 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 [14]. 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 the first step 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 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.

In Cassandra's modified version of Paxos any node can take the role of the proposer and the acceptors are all the participating replicas. The number of acceptors is specified by the serial consistency. Serial consistency has only two levels, LOCAL and and SERIAL_LOCAL. LOCAL requires a quorum of all replicas in the same data center as the coordinator to respond. SERIAL_LOCAL requires a quorum of all replica nodes across all data centers must respond.

Cassandra's modified Paxos consists of four phases. The first phase is the same prepare-phase as original Paxos but the second is a new phase, called the read phase . 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. The third phase is the accept phase of the

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

original Paxos algorithm. The last phase of Cassandra's modified version of Paxos is the commit phase, in which the accepted value is committed to Cassandra storage. These additions to Paxos costs two extra round-trips, resulting in four round-trips instead of two. 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.

3.2.2 | 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. The same data is also written to the memtable, which is a memory cache stored in memory. 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.

3.2.3 | 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. 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. Both the memtable and the row cache is kept in memory, which makes the faster compared to fetching data from the SSTable on disk. If the data is not in the row cache nor the memtable, Cassandra needs to look it up in the correct SSTable. This requires several steps to locate the correct table combined with the fact that the SSTables resides on disk, this option yields much lower latency.

4 | EXPERIMENT DESIGN

In this section we describe the design and methodology of the performance comparison experiments. First, we provide a description and rationale for choice of the cloud platform, followed by a description of how the respective framework were configured to ensure a fair comparison. The test application used in the experiments is also described as well as the evaluation metrics. Finally, we give a brief overview of the five experiments performed.

4.1 | Cloud Platform

Previous work in the area has successfully utilized cloud solutions to deploy Fabric and Cassandra networks. For example, Sukhwani et al. [18] used IBM Bluemix to deploy Fabric and Androulaki et al. [3] used the IBM Cloud. In some papers the authors have chosen to build their own infrastructure using servers for setting up virtual machines, for example, Sousa et al. built their own ordering service [17]. For Cassandra, Amazon EC2 has been used by Kuhlenkamp et al.[12]. There are also examples of when the authors built their own solution, for example the work by Cooper et al. [8].

For the evaluation in this work we analyzed the suitability of four major cloud solutions on the market, Amazon EC2, Microsoft Azure, IBM Cloud and Google Cloud. Each cloud solution was evaluated based on the range of out-of-the-box support for Fabric and Cassandra. Microsoft Azure was chosen based on the available support for the chosen platforms and the ability to run a relatively large number of logical nodes on a single machine (the highest number we achieved was 35). Table 3 shows the specification of the machine on which the tests were run. Both frameworks are setup using Docker with one node per container and all tests are run in the Docker-environment.

TABLE 3 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

4.2 | Configuration

We now proceed to describe how both Fabric and Cassandra are configured in the experiments. This information is provided for the purpose of reproducability. As we describe below, the configuration choices are made for both frameworks to resemble each other as much as possible.

4.2.1 | Hyperledger Fabric

All experiments with a blockchain framework use version 1.1.0 of Hyperledger Fabric, which was the latest version available at the start of our experiments. 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 are connected using a single channel. The policy for endorsement of transactions is a quorum of the 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.

There are two different ordering services implemented in Fabric version 1.1.0, SOLO and a Kafka-based ordering service. SOLO is only meant for testing and not built for a production environment, so we use Kafka in our experiments. 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.

4.2.2 | Cassandra

All experiments with a distributed database use Cassandra version 3.11 and cql version 3.4.4. Cassandra has a tunable replication factor and we configured it to use *N* as replication factor, where *N* is the number of nodes. This is the maximum number of replicas and might seem extreme. 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 by default to allow for large amounts of data to be efficiently processed, but our test application is very lightweight, so each node is restricted to only 64 MB of RAM.

We use LWT transactions in order to enable the Paxos consensus protocol, which provides the same level of fault tolerance as in Hyperledger Fabric. The serial consistency is 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. The serial consistency is used for all LWT operations and overrides the ordinary consistency level.

4.3 | Test Application

The application used for the experiments is a key-value store. The key-value pairs 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 outlined in Section 3.2. 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 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. This seemed to be the most straightforward way to invoke the operations from Docker.

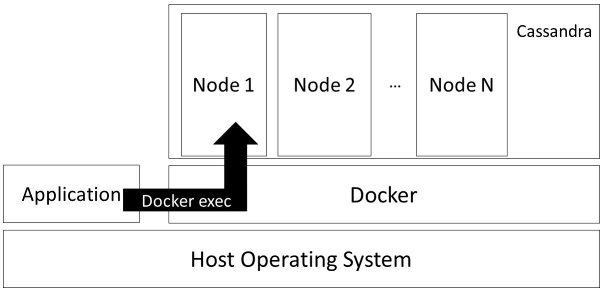

FIGURE 1 Overview of the test setup of Cassandra

In Figure 2 it can be seen how tests for Fabric work on a component-level. 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 2 as peer 1 and peer 2.

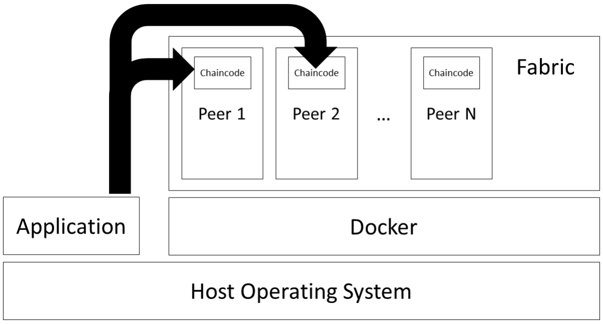

FIGURE 2 Overview of the test setup of Fabric

4.4 | Evaluation Metrics

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

Choice of Latency Metric

The latency of a distributed system can be both measured and defined in a number of ways. The most direct would be to measure 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. However, in distributed systems this is problematic since 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.

A more black-box oriented approach is to measure how fast an operation has an effect on the system. For example, Wada et al. [20] 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.

In this work we measure the round-trip time of an operation. The benefit of this approach is that only the clock at the client end is needed so clock synchronization will not be an issue. The potential drawback of the approach is that some effects in the target system might not have taken place by the time the response is received at the client end. Another problem is the round-trip time is also affected by many other factors such as the network and system software, which we discuss below.

Adjusting for the Overhead

To account for the latency which is not caused by Cassandra or Fabric we model the roundtrip time T_{rt} as the sum of the actual time of an operation within the system T_{op}, and the overhead time T_{oh} which accounts for all the remaining time.

$$T_{\mathrm{rt}} = T_{\mathrm{op}} + T_{\mathrm{oh}} \tag{1}$$

In order to arrive at the value of T_{op}, we must therefore first estimate the overhead time and then subtract this value from the measured round-trip time.

Note that the work done when receiving an insert request before sending confirmation to the client, T_{op}, is different on Cassandra and Fabric. When using LWT in Cassandra, the modified Paxos with four phases needs to finalized 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.

Estimating the Overhead

Since the invocation mechanism for the two frameworks differ (recall Figures 1 and 2), the value of T_{oh} will also differ. We also need to use different methods to estimate T_{oh} for the two frameworks.

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.

For Cassandra, the situation is more complicated. As can be seen in Figure 1 the way to execute a cql-command, for example an `INSERT` or `SELECT` statement, goes through Docker. In this work we 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, as shown in the overview in Figure 3. The test to measure the overhead consisted of measuring the time of executing an empty cql-script. The test was repeated 50 times for each network size.

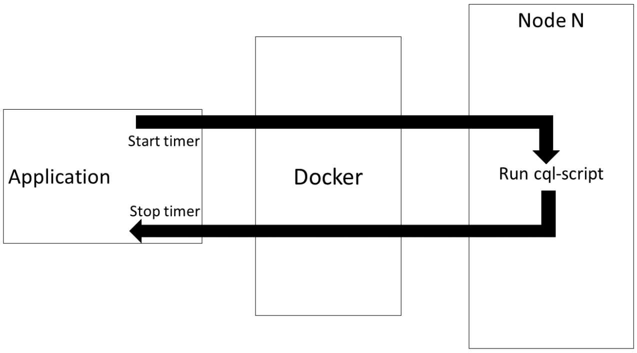

FIGURE 3 Schematic overview of timing measurements for Cassandra

4.5 | Experiment Overview

The results presented in this paper are based on six distinct experiments, listed below.

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
6. Latency for larger network sizes

Each experiment was conducted on both Fabric and Cassandra, configured according to Section 4.2.1 and 4.2.2 respectively. Unless otherwise stated, each run of the experiments contained 50 samples and each experiment was run twice. Networks were brought down between runs to ensure independence 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 paper 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[9]. 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, we were able to run at most 35 nodes (experiment 6).

5 | RESULTS

We structure this section in accordance to the experiment design with one subsection for each experiment, followed by a summary.

In several cases, the results are presented using box plots. Each box represents all data points between the lower and higher quartile. The whiskers represent 95% of all data points.

5.1 | Estimating the Latency Overhead Caused by Docker

The overhead, called T_{oh}, of using the `docker exec` command to run cql-scripts on Cassandra can be found in Figure 4 (note the logarithmic scale). 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 graph also shows the overhead of Docker when using Fabric, which is around 20 ms for all network sizes.

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. This is not a fundamental feature of Cassandra, but a consequence of implementation choices in the setup. 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. It remains to be seen whether similar optimisations can be done to reduce the overhead of Cassandra invocations using Docker.

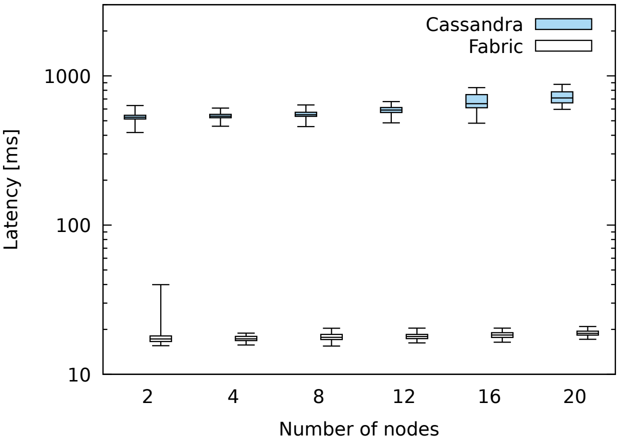

FIGURE 4 The overhead caused by network and system software when using Cassandra and Fabric respectively (log scale)

5.2 | Insert Latency as a Function of Network Size

The purpose of this experiment is to identify how the insert latency is affected by the size of the system. To measure the round-trip time, T_{rt}, 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.

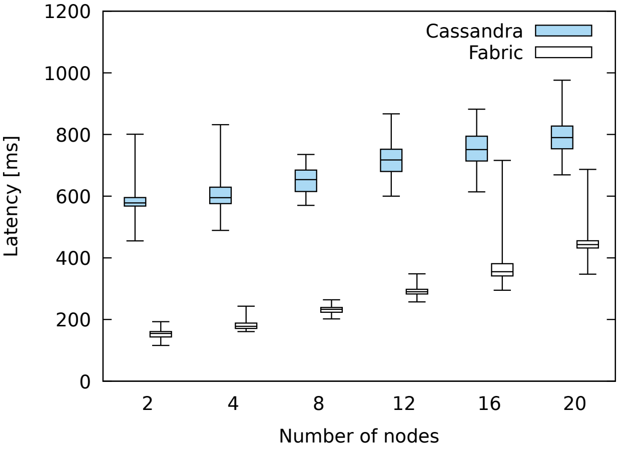

FIGURE 5 The insert latency

The effect of network size on the insert latency for Cassandra and Fabric can be seen in Figure 5. Note that these results are still the raw round-trip time measurements that have not been adjusted for the difference in overhead between the platforms. There are some differences worth pointing out. First of all, Cassandra seems to have a higher latency compared to Fabric. However, as we shall later see, this is mostly due to the difference in overhead. Both system are affected by the increasing number of nodes. In particular, the extreme values for Fabric are much higher for 16 and 20 nodes.

5.3 | Read Latency as a Function of Network Size

The purpose of this experiment is to identify the read latency and how it is affected by the size of the system. 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 measure the round-trip time of a read operation, T_{oh} 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 stated in the previous section. This was repeated for all nodes or peers in the system.

Figure 6 shows the full round-trip time measurements for both Fabric and Cassandra. Fabric has much lower latency than Cassandra (still results are not adjusted for the overhead). The round-trip times for read operations in Cassandra is similar to the insert operations. In Fabric the median read latency is around 40 ms for smaller networks of 2, 4 and 8 peers and around 50 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.

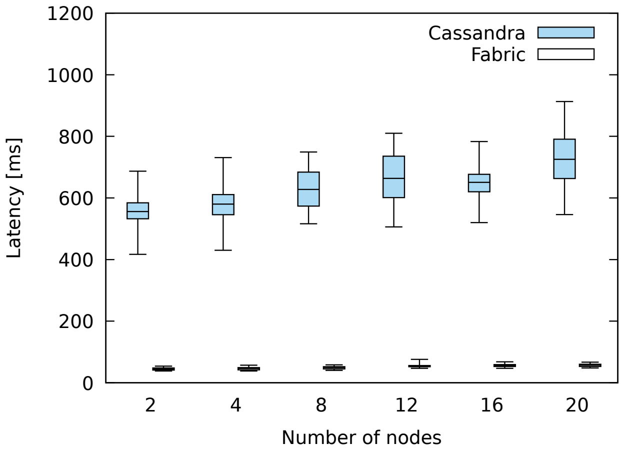

FIGURE 6 The read latency of Cassandra and Fabric

Given the difference in overhead between the two deployments, it is relevant to reconsider these measurements, trying to adjust for the difference in overhead. Note that this is not necessarily a straightforward operation. A new potential source of error is introduced, since subtracting the estimated time T_{oh} from the round-trip time might be overcompensating. Therefore, these results should be interpreted cautiously.

Figure 7 shows the estimated operation time, T_{op} in for both read and insert operations. This value is derived by taking the average round-trip time subtracted with the average estimated overhead from experiment 1. Each case is shown for 2 and 20 nodes respectively.

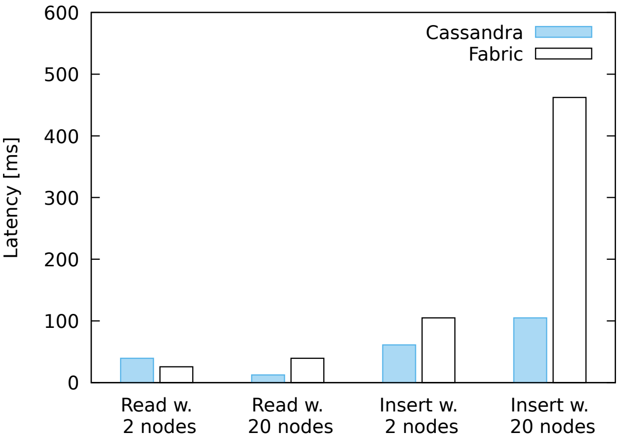

FIGURE 7 Read and insert latencies adjusted for overhead

Interestingly, Cassandra, which seemed to perform so much worse compared to Fabric now outperforms Fabric in all cases except for read operations in very small networks. However, the differences are small for most cases except for insert operations in large networks (20 nodes). Clearly, Fabric does not scale very well, at least not for insert operations. The read operation performs better for Fabric 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.

5.4 | Insert Latency as a Function of Load

The purpose of the next 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 experiment was repeated twice, resulting in 100, 500, 1 000, 1 500 and 2 000 reads respectively.

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 previous experiments only one transaction is performed at a time. This makes setting the `batchSize` to 1 message the most favorable for the ordering service. However, for the load experiment 10 messages correspond to the median number of concurrent transactions and setting the `batchSize` to 10 messages will show how much this parameter affects the overall latency.

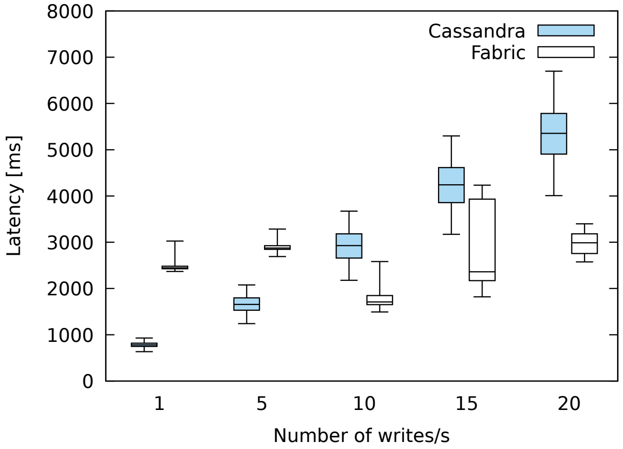

FIGURE 8 The insert latency of Fabric and Cassandra under increasing load

The resulting round-trip times for Fabric and Cassandra can both be seen in Figure 8. Clearly increasing the load has a major impact on the round-trip time for both systems. Cassandra seems be adversely affected by increasing load. One could question if the growing latency would continue for increasing load beyond 20 concurrent inserts per second. We note that Cassandra latency grows linearly. Given that the overhead of the Docker invocation method we have chosen is 90% of the total latency per invocation, we have no reason to believe that this trend will not continue.

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 9 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. There could be seen a similar, but smaller, drop at 20 writes per second.

This behaviour can be attributed to 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.

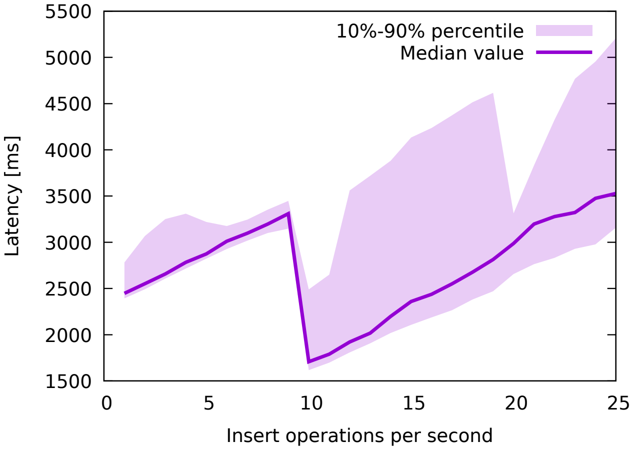

FIGURE 9 The insert latency of Fabric under increasing load - extended

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.

5.5 | 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 (workloads) of insert and read operations. Three different workloads were used in this test, which can be seen in Table 4. The network in these tests has a constant size of 20 nodes or peers. All workloads were run with 100 operations of the least frequently performed operation and were repeated twice. For example in the first row of Table 4 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 operations.

Figure 10 shows the results for the three different workloads. Each bar represents the weighted average latency that correspond to the latency (excluding overhead) for that workload. The weighted average takes account of the latency for read and insert operations and the proportion of each. For example, if T_r is the latency of read operations, and

TABLE 4 Workloads for experiment 5

Name	Fraction read ops.	Fraction insert ops.
Read-intense workload	95%	5%
Balanced workload	50%	50%
Insert-intense workload	5%	95%

T_i is the latency of insert operations, then the result is calculated as $T = rT_r + (1 - r)T_i$, where r is the fraction of read operations.

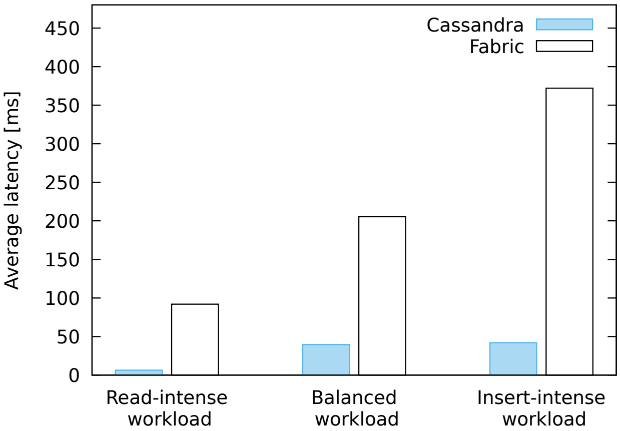

FIGURE 10 The weighted latency of different workloads

In light of the previous experiments, these results are as we expect. Fabric does not perform as well for larger networks, and in particular insert operations are expensive. Cassandra shows a considerable improvement when the workload is dominated by reads. But differently from Fabric, there is very little difference between the balanced workload and the insert-intensive workload.

5.6 | Larger network sizes

In the previous previous experiments the virtual networks were limited to at most 20 nodes, as this was the largest networks that could be run on the available hardware. The purpose of this final experiment is to try to push this bound further by testing on a different more powerful machine that would enable running a larger network size. We deployed a D16s virtual machine with 16 vCPUs, 64GB main memory, and otherwise the same configuration as the previous tests.

With this configuration were were almost able to double the network size from 20 nodes to 35 nodes. Going beyond this number was not possible since the number of internal connections becomes too large. Table 5 shows the averages of the read and insert latencies for two different network sizes (20 and 35 nodes) for Cassandra and Fabric respectively. The results are consistent with the previous experiments, increasing the network size results in generally larger delays, but the increase is still moderate. The largest increase (both relatively and in absolute numbers) is seen for the insert latency of Fabric which goes from 424ms to 593ms as the network is increased from 20 to 35 nodes.

TABLE 5 Average read and insert latencies for Cassandra and Fabric for larger network sizes

	Read latency		Insert latency	
Number of nodes	Cassandra	Fabric	Cassandra	Fabric
20	680ms	50ms	677ms	424ms
35	758ms	57ms	822ms	593ms

One could speculate that with even larger network sizes the difference in insert latency between the two frameworks would decrease further and that Fabric might become slower than Cassandra despite the smaller Docker overhead. However, such performance studies must be done in a truly distributed environment and must consider the effects of the networking layer on the results.

5.7 | Summary

The results in this section provide some important insights, but does perhaps not point to a clear winner. It all very much depends on how much the Docker overhead can be reduced for Cassandra. The raw round-trip time measurements shows Fabric having much lower latency (especially for read operations). However, trying to adjust for the overhead and removing this factor points to Cassandra being the faster framework. In particular for large networks with many insert operations.

It is 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 likely be closer to that of Fabric 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 both systems have almost the same latency.

When it comes to overhead of using Docker, as expected it is clear that Fabric is better optimized than Cassandra.

6 | 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.

We divide this section in four parts. First we briefly present performance studies on permissioned blockchains and distributed databases respectively. Then we discuss how our results compare to what others have reported and how any differences can be understood. Finally, we briefly discuss hybrid solutions.

6.1 | Permissioned Blockchains

Dinh et al.[9] 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. 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.

In later work, Thakkar et al. [19] used benchmarking of the Hyperledger Fabric to understand its specific bottlenecks and propose some mitigations. In particular, based on experiments that changed the configurable parameters of the system such as block size, endorsement policy and so on, they suggested three optimisation policies. They include a membership service provider cache, parallelisation of the validation system chaincode, and bulk read/writes for the multi-version concurrency control. All these optimisations are included in version 1.1 of Fabric that we used in our experiments.

6.2 | 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. [12] 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. [20] evaluate the eventual consistency of 4 different NoSQL databases, including Cassandra

We consider two of these in more detail.

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. [12] 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.

6.3 | Comparison of results

Dinh et al.[9] found that Fabric did not scale beyond 16 nodes. As for latency, their findings are similar to those of this paper. 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. While we have not investigated loads of the same magnitude, the trends are consistent with the ones found by Dinh et al.

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 different setting can greatly effect the latency. The difference can also come from the fact that we run mulitple logical nodes on one physical 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. [12] measure the latency of performing write operations for write-intense workloads. 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 what we found, but still close to the numbers where the overhead is removed. A difference to our work is that we use LWT instead of the standard inserts, LWT is a lot more time-consuming because it establishes linear consistency.

Cooper et al. [8] measure the latency of Cassandra using 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 our work. However, the latency is again lower in their work than what we found. 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 physical machines, instead of one, and allocated 3GB of heap instead of the 65MB.

6.4 | 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[11]. 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[10]. Another example of a similar product is BigchainDB which is a distributed database with added blockchain characteristics like immutability, decentralization and the option to chose permission property per transaction [12]. 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 | DISCUSSION

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. On the other hand this is what most closely resembled the Fabric setup.

The choice of co-locating all the nodes in one single virtual machine might have 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. However, the machine used in this work 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 35 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 described in Section 6, previous work benchmarking both distributed databases and blockchains use networks of similar or smaller size.

Since all our measurements were for single read/write queries, we can only speculate what would happen in scenarios where transactions would consist of more complex queries. We believe the mix of read and writes in a given query, if processed as one batch for a Cassandra invocation, would reduce the relative overhead. However, that mechanism would have to be implemented for Cassandra. For Fabric, we would use the alternative CouchDB which is tailored to invoking more complex queries and thereby exploit the Fabric optimisation for it.

8 | CONCLUSION

To the best of our knowledge this paper is the first work which compares the latency of permissioned blockchains and distributed databases.

When comparing permissioned blockchains to distributed databases it is clear that distributed databases are more mature. 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 built-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.

Despite permissioned blockchains being very young compared to distributed databases, we show that they are comparable to older techniques in terms of latency. In some cases the performance is better, and when factoring in the consistency model, it is likely 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.

Based on the experiments performed, we can see that Fabric pays for relatively 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.

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.
- The system environment software can have a huge impact on performance.

[11] https://chromaway.com/products/postchain/
[12] https://www.bigchaindb.com/

- Does the application require linearized consistency? In that case Fabric could be 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 work 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.

We hope that our work will inspire others to perform similar experiments with other parameters and settings. In particular, benchmarking against similar database platforms (like MongoDB) may be of interest. Clearly, more work is needed to more comprehensively understand what role permissioned blockchains can play in the large database landscape, where each set of requirements is matched by a system tailored to meet exactly those requirements.

9 | ACKNOWLEDGEMENT

This work has been supported by RICS: the research centre on Resilient Information and Control Systems (www.rics.se) financed by Swedish Civil Contingencies Agency (MSB).

References

[1] V. Abramova and J. Bernardino. NoSQL databases: MongoDB vs Cassandra. In Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13. ACM Press, 2013. doi: 10.1145/2494444.2494447.

[2] V. Abramova, J. Bernardino, and P. Furtado. Evaluating Cassandra Scalability with YCSB. In International Conference on Database and Expert Systems Applications, pages 199–207. Springer, Cham, 2014. doi: 10.1007/978-3-319-10085-2_18.

[3] E. Androulaki, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, A. Barger, S. W. Cocco, J. Yellick, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, and G. Laventman. Hyperledger fabric: A Distributed Operating System for Permissioned Blockchains. In Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18. ACM Press, 2018. doi: 10.1145/3190508.3190538.

[4] E. Brewer. A certain freedom. In Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '10. ACM Press, 2010. doi: 10.1145/1835698.1835701.

[5] E. Brewer. CAP twelve years later: How the "rules" have changed. IEEE Computer, 45(2), 2012. doi: 10.1109/MC.2012.37.

[6] C. Cachin. Blockchains and Consensus Protocols: Snake Oil Warning. In 2017 13th European Dependable Computing Conference (EDCC), 2017. doi: 10.1109/EDCC.2017.36.

[7] K. Christidis and M. Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. IEEE Access, 4, 2016. doi: 10.1109/ACCESS.2016.2566339.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10. ACM Press, 2010. doi: 10.1145/1807128.1807152.

[9] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. BLOCKBENCH. In Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17. ACM Press, 2017. doi: 10.1145/3035918.3064033.

[10] J. M. Graglia and C. Mellon. Blockchain and Property in 2018: At the End of the Beginning. Innovations: Technology, Governance, Globalization, 12(1-2), 2018. doi: 10.1162/inov_a_00270.

[11] G. Greenspan. MultiChain Private Blockchain-White Paper, 2015.

[12] J. Kuhlenkamp, M. Klems, and O. Röss. Benchmarking scalability and elasticity of distributed database systems. Proceedings of the VLDB Endowment, 7(12), 2014. doi: 10.14778/2732977.2732995.

[13] A. Lakshman and P. Malik. Cassandra. ACM SIGOPS Operating Systems Review, 44(2), 2010. doi: 10.1145/1773912.1773922.

[14] L. Lamport. Paxos Made Simple. ACM Sigact News, 32(4), 2001. doi: doi:10.1145/568425.568433.

[15] M. E. Peck. Blockchain world - Do you need a blockchain? This chart will tell you if the technology can solve your problem. IEEE Spectrum, 54(10), 2017. doi: 10.1109/MSPEC.2017.8048838.

[16] M. Pisa. Reassessing Expectations for Blockchain and Development. Innovations: Technology, Governance, Globalization, 12(1-2), 2018. doi: 10.1162/inov_a_00269.

[17] J. Sousa, A. Bessani, and M. 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, 2018. doi: 10.1109/DSN.2018.00018.

[18] H. Sukhwani, J. M. Martinez, X. Chang, K. S. Trivedi, and A. Rindos. Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2017. doi: 10.1109/SRDS.2017.36.

[19] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, (MASCOTS), 2018. doi: 10.1109/MASCOTS.2018.00034.

[20] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective. In Proc. Conference on Innovative Data Systems Research (CIDR), volume 11, pages 134–143, 2011.

□