Benchmarking of Distributed Ledger Technology

Bachelor's Thesis

Julien Tinguely

julient@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Dr. Thomas Locher
Prof. Dr. Roger Wattenhofer

October 11, 2019

Acknowledgements

I would especially like to thank my thesis supervisor Thomas Locher for his help, the weekly meetings and for the opportunity to write my bachelor thesis on this subject at ABB's Research Center in Dättwil.

I would like to thank Zahra Farsijani and David Kozhaya for their help with Hyperledger Fabric.

I would like to thank the Distributed Computing Group at ETH under Professor Wattenhofer for hosting my thesis.

I would like to thank my family for their continuous support.

Abstract

Distributed Ledger Technology (DLT) maintains a shared ledger, consistently stored on peers over a network, thanks to consensus protocols. This technology often enables smart contracts, which are increasingly being used and promise a revolution in the way assets are exchanged, as well in industry as in government [1].

One state machine replication framework and two distributed ledgers are analyzed here: BFTSMaRt, Corda R3, and Hyperledger Fabric. BFTSMaRT is a replication library that offers a good basis for comparison to new distributed ledgers implementing smart contracts, such as the two rivals Corda and Fabric.

Throughput in term of transactions per second is the main metric to evaluate the performance of such technologies. Measuring it on a cluster of servers gives results close to the performance in real-world deployments. The work accomplished in the thesis can be extended to perform measurements for a wide range of uses cases.

Smart contracts have a cost. BFTSMaRt's simplicity offers high throughput but does not support smart contracts. Different implementations, workloads, and number of clients often imply different performances. Often trade-offs between implementation's architecture and throughput are decisive before starting new projects using this technology. For that reason, benchmarking is an important factor to take into consideration.

List of Figures

2.1 Generic DLT Building Blocks 5
2.2 BFTSMaRt replica staged message processing [2] 8
2.3 Corda transaction's flow 12
2.4 Corda node's architecture 13
2.5 Fabric's architecture 15
2.6 Fabric transaction's flow 16

3.1 Trinity cluster scheme 21
3.2 Caliper's flow diagram 24
3.3 Corda's Swarm configuration 29
3.4 caliper-corda flow diagram 31

4.1 BFTSMaRt: benchmarking setup 35
4.2 BFTSMaRt: 2 servers, no signature 36
4.3 BFTSMaRt: 4 servers, no signature 36
4.4 BFTSMaRt: 8 servers, no signature 36
4.5 BFTSMaRt: 16 servers, no signature 36
4.6 BFTSMaRt: 2 servers, with signatures 37
4.7 BFTSMaRt: 4 servers, with signatures 37
4.8 BFTSMaRt: 8 servers, with signatures 37
4.9 BFTSMaRt: 16 servers, with signatures 37
4.10 BFTSMaRt: effects due to number of servers 38
4.11 BFTSMaRt: time per phase 39
4.12 BFTSMaRt: response time distribution 39
4.13 Corda: benchmarking setup 40
4.14 Corda: increment issue 41
4.15 Corda: increment 41
4.16 Corda: cash issue 41

4.17 Corda: payment 41
4.18 Fabric: benchmarking setup 43
4.19 Fabric: increment on 1 organization 44
4.20 Fabric: query on 1 organization 44
4.21 Fabric: issue on 1 organization 44
4.22 Fabric: increment on 2 organizations 45
4.23 Fabric: query on 2 organizations 45
4.24 Fabric: issue on 2 organizations 45

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Distributed Ledgers 1

1.2 Benchmarking of Distributed Ledger Technology 2

2 Distributed Ledger Technologies **3**

2.1 Background 4

2.2 BFTSMaRt 6

2.3 Corda 10

2.4 Fabric 14

3 Methodology **17**

3.1 Related Work 17

3.2 Current Work 18

3.3 Benchmarking 22

3.3.1 A Generic Benchmarking Tool 22

3.3.2 BFTSMaRt 26

3.3.3 Corda 28

3.3.4 Fabric 32

4 Measurements **34**

4.1 Results 34

4.1.1 BFTSMaRt 34

4.1.2 Corda 40

4.1.3 Fabric 43

5 Conclusion **46**

Bibliography **48**

A Annexes **A-1**

A.1 BFTSMaRt Benchmarks bash script A-1

A.1.1 setup_trinity_bftsmart.sh A-1

A.1.2 launchbenchmarks.sh A-1

A.2 Increment example A-3

Chapter 1

Introduction

1.1 Distributed Ledgers

Ledgers[1] have been used since ancient times to record assets, transactions, and contracts. They evolved in time being stored on papyrus, paper, and nowadays commonly in databases. Notaries, states, banks, and companies are currently considered as trusted third parties that certify the validity of their records. They are therefore centralized systems prone to attacks or loss of records [1].

Can we make ledgers more efficient, more secure, and remove the overhead of a unique (sometimes physical) third party?

The emergence of Bitcoin in 2008 gave a new impetus to the study of distributed systems. The almost 50 years of studies that leads to the reliable blockchain technology[2] hides huge work on consensus algorithms, consistent replications, and security. Bitcoin itself leads to the emergence of smart contracts and gave birth to the Distributed Ledger Technology (DLT).

Smart contracts aim to automate transactions between parties within a peer-to-peer network by authenticating them in a distributed fashion. It moves the disconnected trusted third party into the network's participants themselves.

DLTs are split into two categories: permisionless or public DLT, which anyone can join (Bitcoin, Ethereum, ...), and permissioned or private DLT, which is restricted to pre-defined members (Corda, Fabric, Tendermint, ...). The first is more suited for cryptocurrencies as anyone using it should be able to take part in its network. The second is mostly designed for companies or governments. For instance, a company could make use of it to automate some of their processes by implementing smart contracts with their suppliers or clients in a secure, efficient, and private form.

Some are convinced that the deployment of Distributed Ledgers in our daily life is going to radically change our society. This technology will most likely

[1] synonyms: account or record books.

[2] The blockchain is a type of distributed ledger that is structured in a sequence of blocks and is provably secure.

replace some well established centralized ledgers within the next few decades, and for that reason, be one of the biggest innovation of the 21st century [3].

1.2 Benchmarking of Distributed Ledger Technology

The thesis focuses on measuring the performance of private distributed ledgers. It aims to give the reader a short introduction, an insight into how performance was measured and throughput expectations of BFTSMaRt, Corda, and Fabric.

Distributed ledgers' main ingredients are: servers, clients, a network, an application, and transactions. Chapter 2 defines these components and their relations in the first section. The three following sections introduce the above mentioned replication framework and distributed ledgers and explain how these components are mapped into their architectures.

Benchmarking is a vague task. Performance of networks systems can be measured in all OSI layers[3]. As one goal of this thesis is to compare distributed ledgers, it only considers the application level. Here, performance is defined by its throughput in term of transactions per second, its transaction response time, its transactions success and failure rate, between a client and servers. It omits security aspects. The core of the work is presented in Chapter 3 and aims to help the reader to reproduce and create new benchmarks.

As interest for this technology is growing, it is important to compare the different implementations in term of performance and platform-specific limitations. Chapter 4 gives some measurement results for particular examples as well as their interpretation.

The report is structured as follows: Chapter 2 introduces the analyzed distributed ledgers technologies by presenting their general key ideas. Chapter 3 presents the benchmarking methodology as well as how the benchmarks were produced. Chapter 4 presents some benchmarking results, generated with the tool Caliper, which was used and extended throughout the thesis. The results are discussed and interpreted in Chapter 5, which concludes the report.

[3]An Open System Interconnection classifies a system's components into layers.

CHAPTER 2

Distributed Ledger Technologies

Chapter 2 first gives definitions in order to understand what a distributed ledger is, what it has to accomplish, and how servers, clients, consensus protocol, and smart contracts are related. It then briefly presents in the following sections the three concrete implementations of distributed systems, that are analyzed:

- BFTSMaRt, a byzantine fault tolerant replication framework implemented in Java,
- Corda, a DLT using JVM[1] and originally designed for finance, implementing an order-execute architecture, and
- Fabric, a DLT using containers and known for its modularity, implementing an execute-order-validate architecture.

[1] Java Virtual Machine

2.1 Background

Key Concepts

For more information about distributed systems and networking in general, please read [4] and [5].

Definition 2.1. Peer

Peers can be both client and server. They are often both in DLT. They are sometimes referred to as parties or nodes in some documentations.

Definition 2.2. Peer-to-Peer Architecture

It is an architecture where there is no special party that provides a service or manages the network resources. All responsibilities are uniformly divided among the peers.

Definition 2.3. Consensus

A consensus algorithm is a protocol that aims to achieve agreement on shared data in a distributed system. There exist many consensus algorithms such as Paxos, Raft, PBFT (each having their pros and cons). It must meet three properties: termination, integrity, and agreement.

Definition 2.4. Transaction

A DLT's transaction has parallels to real-life transactions. It aims to update the distributed ledger state by changing the ledger's content. A transaction could be for example: a financial transaction, peer A sending money to peer B, which changes the shared state such as all concerned peers are consistent with the update; a diamond identity issuing transaction, which registers the diamond ID and authenticity; a dummy transaction, which increments a number such that all peers store the same number.

Definition 2.5. Smart Contract

A smart contract is a protocol intended to verify and improve performance of a contract. Smart contracts allow:

- the issuance of tangible transactions without third parties,
- controlled access to the ledger,
- simple and structured encapsulation of data, and
- participants to execute certain aspects of a transaction automatically.

Definition 2.6. Distributed Ledger

A distributed ledger is a database recording transactions that is shared, replicated, and synchronized by members of a decentralized network of peers, having the following features:

- its peers are bound together within a decentralized network and therefore form a distributed system. They are replicated servers connected in a peer-to-peer architecture, running the same application waiting on transactions to occur,
- there is no centralized data storage or unique administrator. Members of the network agree together on the database state by consensus,
- every transaction in the distributed ledger has a time-stamp and a unique cryptographic signature in order to have an immutable history of all transactions.
- they can implement smart contracts, and
- in contrast to centralized ledgers that are prone to cyber-attacks, an attacker would usually need to modify more than one copy of the network's quorum for the attack to be successful. This makes the DLT secure.

Definition 2.7. Blockchain

A blockchain is a kind of distributed ledger. Data on a blockchain is organized in blocks that are linked to one another and secured using cryptography.

Definition 2.8. Generic DLT Building Blocks

Figure 2.1 gives a generic view of a DLT building blocks. Level 3 defines the mechanism and the location of code execution and its data storage, level 2 the protocol for consensus, and level 1 the ledger data structure, how the distributed ledger is structured and linked at a physical level [6].

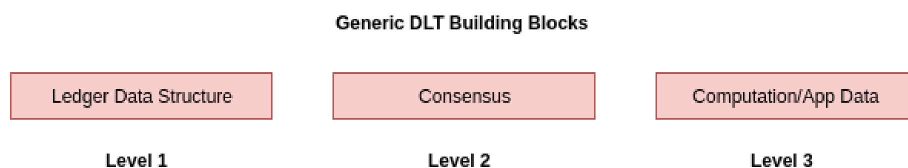

Figure 2.1: Generic DLT Building Blocks

2.2 BFTSMaRt

Overview

BFTSMaRt is a Byzantine fault-tolerant state machine replication framework[2] from 2008. It implements a simplified version of the BFT consensus algorithm. It uses a state machine replication protocol which is similar to Paxos [2].

BFTSMaRt uses Java. This choice is justified by its popularity. The BFTS-MaRt community nevertheless gives strong importance on performances [7]. The library is open source, does not implement smart contracts, and also aims to be integrated in other DLT such as Fabric.

Key Concepts

To learn more on BFTSMaRt, please read [2].

Replica

A replica is a peer running the BFTSMaRt's library and an arbitrary application. Replicas are started in different shell and can be located on different computer systems. A replica is running within a server and is started in the root directory with the command:

bftsmart.demo.*APPLICATION_ NAME.APPLICATION_ NAME*Server -*ID*

Client

A client is also a peer running the BFTSMaRt's library and an arbitrary application. Replicas and clients communicate via a service proxy that takes care of broadcasting messages. It is started in the root directory with the command:

bftsmart.demo.*APPLICATION_ NAME.APPLICATION_ NAME*Client -*ID* -*ARGUMENTS*

Consensus

System machine replication is a technique where an arbitrary number of clients issue commands to a set of replicas. These replicas implement a stateful service that changes its state after processing the client commands and sends replies to the client.

[2]A general method for implementing a fault-tolerant service by replicating servers and co-ordinating client interactions with server replicas.

The main purpose is to maintain consistency among replicas. The service is replicated completely and consistently on each replica, which means:

- replicas only apply deterministic changes to state,
- all replicas start with the same state, and
- all replicas execute the same sequence of operations.

The first two requirements are met without any special library. The last one requires communication. It requires the execution of a complex consensus protocol to ensure commands are executed in the same order across all replicas.

The consensus protocol is an optimized version of BFT [7] and is therefore designed to tolerate Byzantine faults[3]. Byzantine faults can be due to: software bugs, skilled attacker who wants to gain control of replicas. The total number of Byzantine replicas f cannot be larger than $f \leq \frac{N-1}{3}$, with N being the total number of replicas in the network.

The network always contains a leader protocol replica that is not chosen by the client or anyone that administrates the system. The protocol votes for a new leader if a request timeout is triggered, the leader stops behaving as expected, network delays, or more requests are queued than the system is able to process.

Transaction Flow (Figure 2.2)

1. Unordered requests, which are usually used for read-only commands, are directly delivered to the service implementation.
2. The client manager verifies the request integrity and adds them to the respective client's queue.
3. The proposer thread is responsible for assembling a batch of requests and transmitting the PROPOSE message of the consensus protocol.
4. The batch is then filled with pending requests until: (a) its size either reaches a limit defined in the configuration file; or (b) it has no request left in the queue. This thread is only active at the leader replica.
5. Every message to be sent by one replica to another is put on the out queue from which a sender thread will get the message, serialize it, produce a MAC to be attached to the message and send it using TCP sockets. It is known as the ACCEPT phase.
6. When a decision is reached on a replica, the decided batch is put on the decided queue. The WRITE phase starts.

[3]It can be any deviation from the expected behavior of a replica.

7. The delivery thread is responsible for getting batches from this queue, deserialize all requests from the batch, remove them from the respective client queues and mark the current consensus as finalized.

8. The request timer thread is periodically activated to verify if some request remained for more than a pre-defined time on the pending requests queue.

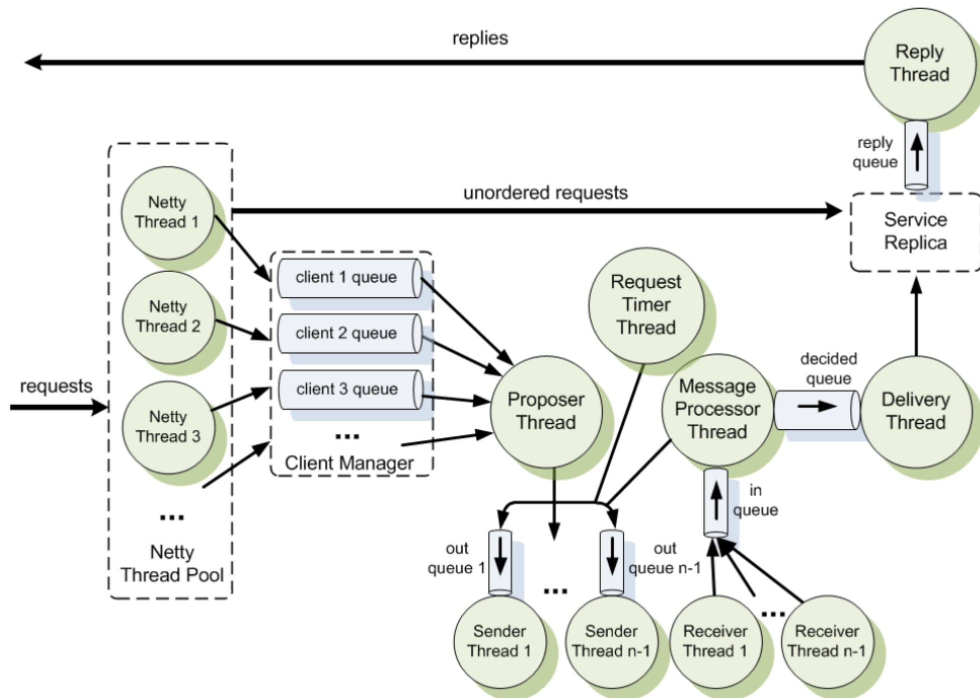

Figure 2.2: BFTSMaRt replica staged message processing [2]

Software Requirements

1. The reader needs to have Java and ant installed. (Eclipse has both)
2. **Download the zip file** from the official website[4]: library-1.1-beta
3. Unzip library-1.1-beta.zip, go into /library-1.1-beta
4. Configure the file "config/hosts.config" by writing the id, the IP address and the port for each server
5. Configure the file "config/system.config" by changing the system.servers.num and the inital system view (with server ids)
6. Always remove the "config/currentView" before each run, if the network has been reconfigured.
7. Be sure that the private and public keys exist for the defined server (it will take the prefix "privatekey" and adds the server id as key name).
8. Start each server in a different shell:
   ./runscripts/smartrun.sh bftsmart.demo .counter.CounterServer -SERVER_ ID
9. Start a client in a different shell:
   ./runscripts/smartrun.sh bftsmart.demo.counter.CounterClient -Process_ ID -Increment -Number_ Ops
10. Any change must be compiled with *ant -buildfile build.xml*

[4]http://bft-smart.github.io/library/

2.3 Corda

Overview

Corda was launched by the company R3 in November 2016 and its architecture is mainly influenced by the financial industry. It claims good business-logic models between firms and economic actors.

It is implemented in Kotlin and therefore runs on JVM. It offers an Enterprise subscription which implements many useful functionalities. Each peer represents a JVM runtime-environment and hosts Corda services and smart contracts applications named Cordapps.

Like Bitcoin, it uses the UXTO[5] model: Corda's transaction create immutable states that are consumed by another transaction creating unspent transactions or unconsumed states.

Key Concepts

These two papers offer a good introduction to Corda: [8] and [9].

Node

A node is a peer that hosts Corda services and smart contracts applications named Cordapps, generated with the help of gradle[6]. Each node runs on a server and its identity is stored on a shared network map service. It has the following structure:

```
./
├─ corda.jar  # The core Corda libraries
├─ node.conf  # The node's configuration files
├─ cordapps/  # The CorDapp JARs installed on the node
├─ certificates/  # The node's certificates
├─ brokers/  # Stores buffered RPC messages
├─ artemis/  # Stores buffered P2P messages
└─ drivers/  # Stores other drivers
```

[5]UXTO stands for "unspent transaction output".
[6]Gradle is an open-source build-automation system.

Notary

A notary is a special node that prevents double-spends and at the same time acts as a time-stamping authority. It can also validate transactions in which case it is called a validating notary. A cluster of notaries enables the consensus algorithm to run, but this functionality is unfortunately only available on Corda's Enterprise version.

Ledger

Each node maintains its own ledger. The ledger is then subjective from each node's perspective. Two nodes that are in relation are guaranteed to see the same facts they share on the ledger.

State

States represent on-ledger facts and are immutable. States are created in a bitcoin fashion by marking consumed states as historic and creating updated states marked as unconsumed.

Vault

The vault stores all consumed and unconsumed states.

Contracts

Corda's contracts are meant to check that a transaction is valid. It would directly reject it if a contract's clause is not satisfied. A contract execution is deterministic and its acceptance is based on the transaction contents alone. It must be verified by all validating parties. A peer is under no obligation to sign a transaction just because it is contractually valid.

Flows

Flows automate the process of a transaction as a whole. It takes care of communication and ledger updates. They are closely related to transactions. There are no broadcast messages.

Consensus

Consensus runs in a notary cluster. A transaction must achieve both validity and uniqueness consensus to be valid. The validity consensus requires contractual

validity and uniqueness consensus prevents double-spends.

Transaction Flow

A transaction is a proposal to update the ledger and will only be committed if it does not contain double-spends, it is contractually valid, and signed by the required peers.

Issuing transactions create new states in the network and committing should be seen as modifying states. For that purpose each transaction has arbitrarily many input states and output states.

Figure 2.3 shows the life-cycle of a transaction. The consensus between notaries occurs at step 9.

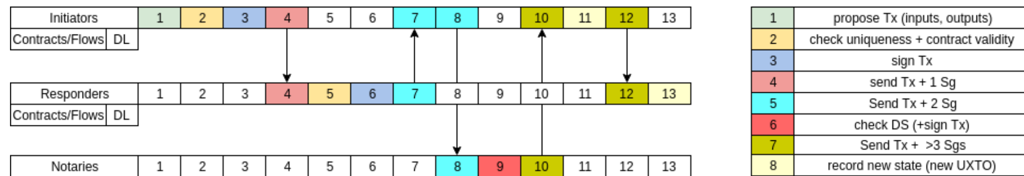

Figure 2.3: Corda transaction's flow

Client

A client of the network can participate in the network by connecting to a node. It connects to a node either via RPC[7], an API, or directly through its shell via SSH[8]. Once connected it can initiate flows (FLOW START), thus issue or commit transactions in the distributed system in a secure way. (figure 2.4)

[7]Remote Procedure Call
[8]Secure SHell

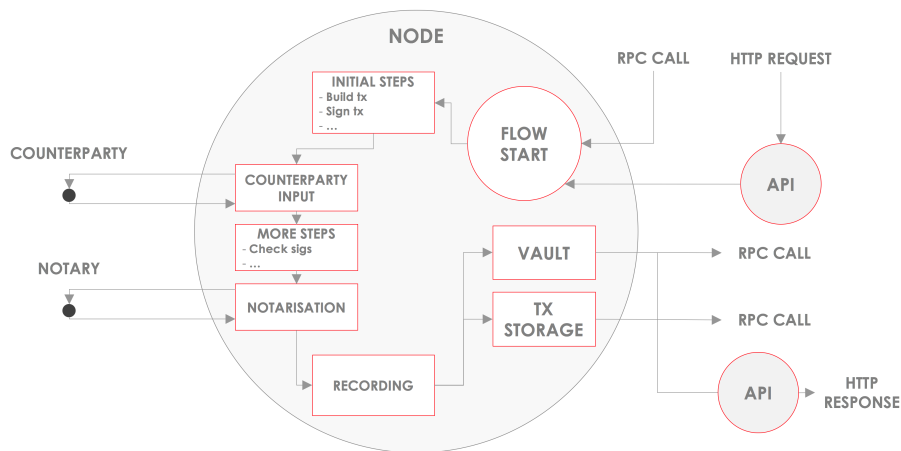

Figure 2.4: Corda node's architecture

Software Requirements

Please refer to the following link for the installation[9].

1. Install all dependencies required by Corda,
   (a) Intellij
   (b) gradle should be wrapped in Corda's code
2. Clone and use the *corda-template* or an existing Corda's example on your machine,
3. Implement flows in the /workflows subfolder, contracts and states in the /contracts subfolder,
4. Use JUnit to test the application locally,
5. Configure the nodes in build.gradle and generate the nodes under build/nodes (either in Cordform or Dockerform) with the help of gradle,
6. Deploy the nodes on a network, and
7. Connect to a node to participate in the network created (see section 3.3.4).

[9] https://docs.corda.net/getting-set-up.html

2.4 Fabric

Overview

Hyperledger Fabric is an open-source system for deploying and operating permissioned blockchains. The project started in December 2015 by the Linux Foundation under the impulses of IBM, Intel, Fujitsu, and JP Morgan.

It is designed to map the complexity existing across the economic ecosystem. For that reason Fabric is modular: it can run different consensus algorithms and databases, and it can be used for different distributed applications and support different programming languages for its smart contracts: Go, Java, and Javascript.

Fabric is implemented in Golang and uses container technology[10] to deploy its network to deliver a enterprise-ready network security, scalability, interoperability, and portability.

Key Concepts

If the reader wants to read more about Fabric: [10] and [11].

Chaincode

A Chaincode is Fabric's smart contract. It is a software that contains the business logic of the system. A Chaincode is invoked when an application needs to interact with the ledger.

Ledger

The shared ledger records the state and ownership of assets. The ledger consists of the world state that describes the state of the ledger at a given point in time and the blockchain which logs the transaction history.

Peers

Peers are fundamental in the network because each one hosts ledgers and smart contracts. A peer executes Chaincode, accesses ledger data, and are the clients' entry points. Some peers can be endorsers which may specify an endorsement policy, which defines the necessary and sufficient conditions for a valid transaction.

[10] A container is a standard unit of software that packages up code and all its dependencies so the application can be ported quickly and reliably from one computing environment to another.

Channel

A Channels allows a group of peers to create a separate ledger of transactions, accessible only by members.

Organization

Peer nodes are owned by different organizations. Organizations manage their own peers with the help of a Membership Service Provider that assigns an identity (digital certificate) to each organization's peer. Peers from different organizations can be on the same channel. Organizations can be as big as a multinational corporation or as small as a coffee shop.

Orderer

Orderers are peers that have the following roles:

- they manage the ordering service. The ordering service packages transactions into blocks to be delivered to peers on a channel. It guarantees the transaction delivery in the network. It communicates with peers and endorsing peers, and
- they maintain the list of organizations that are allowed to create channels.

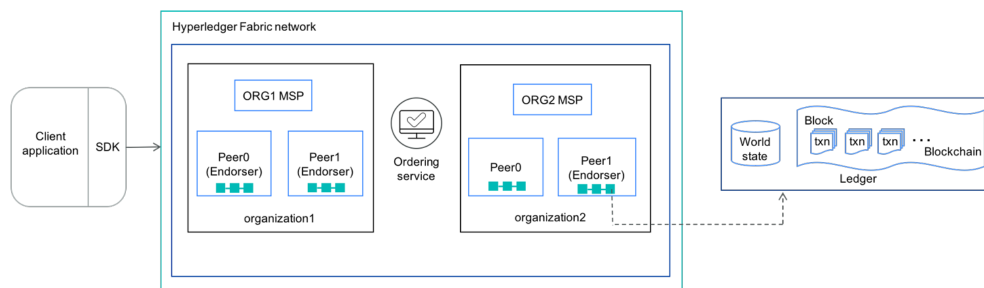

Figure 2.5: Fabric's architecture

Consensus

The consensus occurs within the Ordering Service and involves Orderers. It occurs at stage 6 in Figure 2.6. Fabric offers three main consensus algorithms: Solo (one Orderer), Raft, and Kafka.

Transaction Flow (see Figure 2.6)

1. A client signs and sends a transaction to endorsers (1),
2. its endorsing peers verify the signature and simulate the proposal using the specified Chaincode (2),
3. proposal responses are inspected by the client (3),
4. the client assembles all endorsements into a transaction and submits it to the Ordering service (4-5),
5. the new transaction block is validated by Orderers and committed (6-7), and
6. all peers' ledgers update their local ledgers (8).

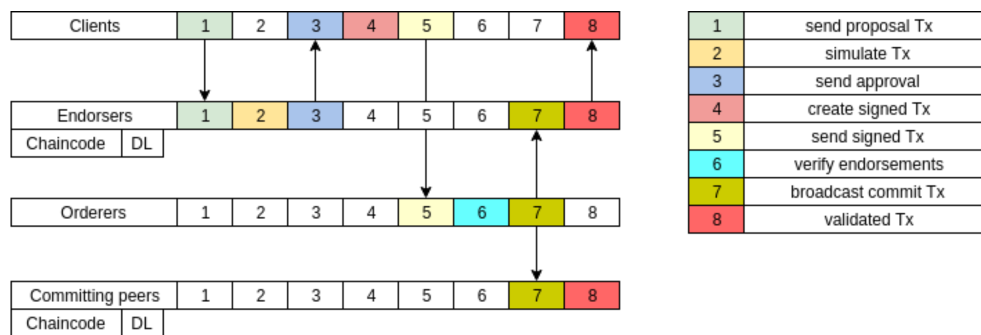

Figure 2.6: Fabric transaction's flow

Software Requirements

See references on the different installations on the following link[11].

1. Install curl: *sudo apt-get install curl.*
2. Go to Docker install on ubuntu and follow the guidelines on installing, from package (sudo dpkg -l ...).
3. Install docker compose 1.23.2.
4. Install Golang 1.12 and set its Path (export GOPATH=$HOME/go in /.bashrc).
5. Install NodeJs 8.11.X and npm 6.4.1.
6. Install Hyperledger Fabric 1.4 via Curl (see on website).
7. Implement the first chaincodes (see Section 3.3.4).

[11] https://hyperledger-fabric.readthedocs.io/en/release-1.4/getting_started.html

Chapter 3

Methodology

In the following paragraphs, the benchmarking methodology is presented. The reader will get an idea about the related work already accomplished in this field, the assumptions taken through the thesis, and how the benchmarks were created.

3.1 Related Work

Some papers and articles aim to give performance results of specific distributed ledgers: Fabric [12], Corda [13]. and BFTSMaRt [2].

Only few theses and papers have been published about comparing DLT's performance. Most of them concentrate on one or two parameters. For instance, both Ethereum and Fabric have been compared in terms of workloads [14]. The main reason is that it is not an easy task; different implementations often require different configurations and parameters. It makes comparison difficult and sometimes even impossible: a Notary is totally different from an Orderer.

Often there is a gap between the paper's results and the ones the reader might get, because some do optimization and benchmarking at the same time [13] [15], which does not give the same results when starting from scratch. They seem to be more closely related to some "marketing benchmarks" and sometimes even measure the throughput from the network perspective (how many transactions occur per second in the network). Thus more nodes in the network implies higher throughput.

Sometimes benchmarks are also a bit irrelevant as they run large networks, locally on a laptop, thus not considering the main power of this technology: its ability to distribute resources.

The Hyperledger foundation started the Hyperledger Caliper project in early 2018. This platform gathers all benchmarking tools for the Hyperledger technologies. They have the ambition to become a generic tool, extendable for any DLTs. [16] and, more importantly, to bring developers together around this common goal.

3.2 Current Work

Overview

In this thesis two aspects are studied: the integration of BFTSMaRt, Corda, and Fabric into a unique benchmarking tool and comparing them to one another. The trade-offs are discussed based on measurements and architecture designs.

To distinguish this work from what has been done, a goal is to compare them on a fair basis without optimizing them. As we cannot compare apples to oranges, the decision was to take these platforms without modifications and measure how many transactions a client can execute per second for each platform. Therefore, throughput and response time are the main metrics.

Deploying peers on a cluster requires time and some understanding in the technology itself. Another goal of the thesis is therefore to summarize the knowledge acquired to give some useful tricks to the reader.

For these reasons, the decision taken was not to create a separate project, but to fork Caliper, an existing project[1] which is active in the field, with the hope that the code could help others at some point.

Choice of Distributed Ledgers

In this project three different open source projects are analyzed: BFTSMaRt (2008), Corda (2016), and Hyperledger Fabric (2015).

Originally, Tendermint was supposed to be benchmarked as well. Tendermint was reconsidered and replaced by BFTSMaRt. The reason is that BFTSMaRt also implements the BFT consensus protocol, it is 6 years older, and an adapter is being developed to be run on top of Fabric.

Moreover, benchmarking a consensus framework gives interesting results about the time spent in the networking and consensus layer. It is therefore a good project to be compared to DLT projects. It is known for its efficient BFT implementation and has also already been used in many projects.[2]

Corda and Fabric are the main private DLTs. They are two direct rivals proposing an open source implementation of DLT for enterprises and offering end-to-end smart contract delivery within a network. Corda has an optimized Entreprise version as well, which is partially analyzed here.

[1] https://github.com/jutinguely/caliper
[2] https://github.com/bft-smart/library/wiki/Sponsors

Metrics

The metrics used underneath are good indicators for measuring performance of distributed systems:

- Throughput: $trp = \frac{|\{transactions\}|}{end_{bench} - start_{bench}}$ in $[transactions\ per\ second] = [tps]$
- Response time: $rsp = end_{tx} - start_{tx}$ in $[ns]$ [3]
  - Average response time: $\overline{rsp} = \frac{\sum rsp}{|\{rsp\}|}$,
  - Maximum response time: $rsp_{max} = \max\{rsp\}$, and
  - Minimum response time: $rsp_{min} = \min\{rsp\}$
- Standard deviation (resp. time): $std_{rsp} = \sqrt{\frac{1}{N}\sum_{i=0}^{n}(rsp_i - \overline{rsp})^2}$ in $[ns]$
- Standard deviation (throughput): $std_{trp} = trp_{avg}\frac{std_{rsp}}{\overline{rsp}}$ in $[tps]$
- Latency: time taken for a task execution in some parts (consensus latency, communication latency, ...) in $[ns]$
- Fail/Success rate: number of transactions that are not/are accepted by the system in percent

with $start_{bench}$ being the time measured at the beginning of the benchmark and end_{bench} at the end.[4]

Similarly $start_{tx}$ the time measured when the client sends a transaction (tx) to the network and end_{tx} when it receives a reply.

Setup

Local

A computer's performance is bound by its own processing power. Running a network locally enables testers to compare and discover some connections between parameters, but does not show how the network scales and the expected throughput. Local benchmarks can therefore only be relevant if there is one validating authority: one server (BFTSMaRt), one notary (Corda) and one orderer (Fabric).

[3] $1[s] = 10^3[ms] = 10^9[ns]$

[4] In Java, $start_{bench}$ is measured by the method *System.nanoTime():Long* and contains enough bytes to store a large time interval as *Long.MAX_ VALUE = 9'223'372'036'854'775'807.*

Cluster

A cluster consists of interconnected servers communicating together. A cluster's computational power strongly depends on the number of computers it contains. Clusters enable the deployment of DLT, as any distributed systems needs to have distributed hardware (servers) to work.

Note that a lot of deployed distributed systems like VISANet, the credit card network from Visa, build huge clusters to take advantage of replication and its high computational power. They go even further to get the best performance: they have machines assigned to specified tasks. This concept of dividing a node's functionality is known as sharding[5].

The cluster used for benchmarking is composed of 50 nodes fully connected and located at ABB Research Center. A maximal number of 17 servers was used for the thesis (see Figure 3.1). Machines' names are separated into Cheetahs and Cougars. The following hardware are used: *Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz* for Cougars and *Intel(R) Core(TM) i5-3470 CPU @ 2.70GHz* for Cheetahs. Both have 4 cores CPUs. The average round-trip ping between any two machines is about 0.312 [ms].[6]

[5]Fragmenting nodes' functionalities, more information on https://www.computerworld.com/article/3336187/sharding-what-it-is-and-why-so-many-blockchain-protocols-rely-on-it.html

[6]Value averaged over 100 measures between 4 machines.

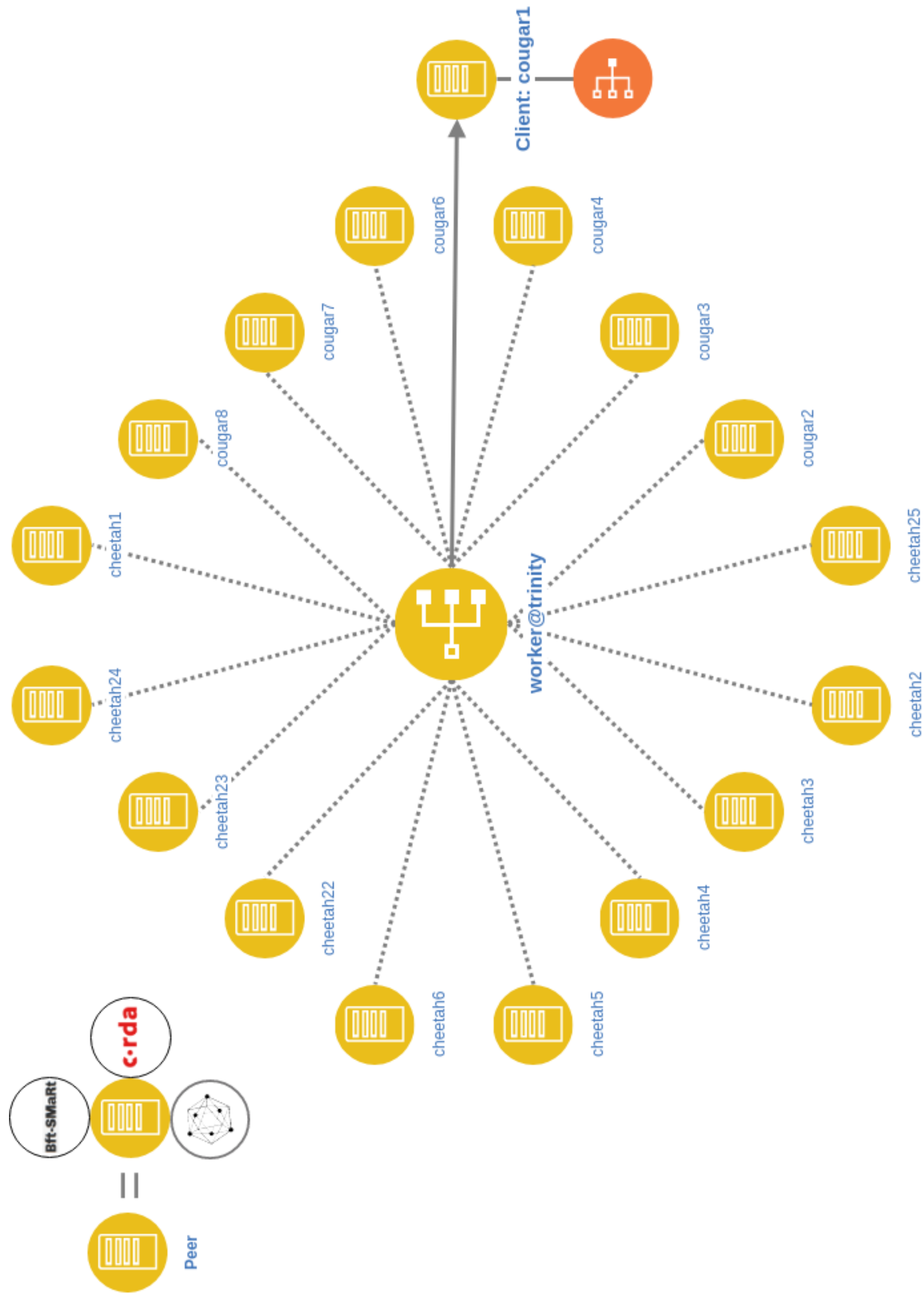

Figure 3.1: Trinity cluster scheme

3.3 Benchmarking

The following section describes in details how the benchmarks were done for each DLT solution and gives useful recipes to deploy them.

The reader[7] will find implementations of modules used to benchmark the different DLT on the forked repository: https://github.com/jutinguely/caliper.

The modules have been implemented in order to integrate Caliper as adapters and are for that reason named caliper-MODULE. Be aware that some adjustments might be needed to reproduce the steps as DLT and the technologies used evolve quickly.

3.3.1 A Generic Benchmarking Tool

Overview

The goal is to collect interesting information, such as response time and throughput, from a distributed ledger running a particular application, in order to compare its performance range with others. We want to know how many transactions a client can execute per second.

To accomplish it, we need:

- to deploy a network of interconnected servers and configure them to run the distributed ledger and (smart contract) application in question, and
- to have clients sending transactions to servers at a high rate during a time interval to record the measured data.

The concrete servers and clients should therefore implement the following abstractions:

[7] As some parts include tutorials, from now on until the end of Chapter 3, the third person might be replace by the second person.

Algorithm 1 i^{th} Server's Process

```
1: setup running environment, load (smart contract) application
2: connect to peers, elect leader if necessary, maintain secure connection
3: while server is up do
4:    process incoming transactions (consensus, local execution)
5:    if connection is interrupted then
6:       goto 2
7:    end if
8: end while
```

Algorithm 2 Client's Process [n: number of transactions, t: list<number of threads>, s: list<request size>, ? p: list<other parameter>]

```
1: setup running environment, load application
2: for each i in t do
3:    create i threads
4:    for each thread do
5:       for each size in s do
6:          for each i in [0..n/i] do
7:             start timer
8:             send(transaction) to a proxy that delivers them to servers
9:             end timer and save it in list<time>
10:         end for
11:      end for
12:   end for
13:   compute metrics, save benchmark in results
14:   reset environment, rest
15: end for
16: return results
```

Caliper

Caliper is a blockchain performance benchmark framework, which allows users to test different blockchain solutions with predefined use cases and get a set of performance test results. It was started by Hyperledger in early 2018 and measures the following indicators:

- Transaction throughput
- Transaction latency (minimum, maximum, average, percentile)
- Success rate
- Resource consumption (CPU, Memory, Network IO, ...)

It currently supports the Hyperledger blockchain projects: Burrow, Composer, Fabric, Iroha, and Sawtooth.

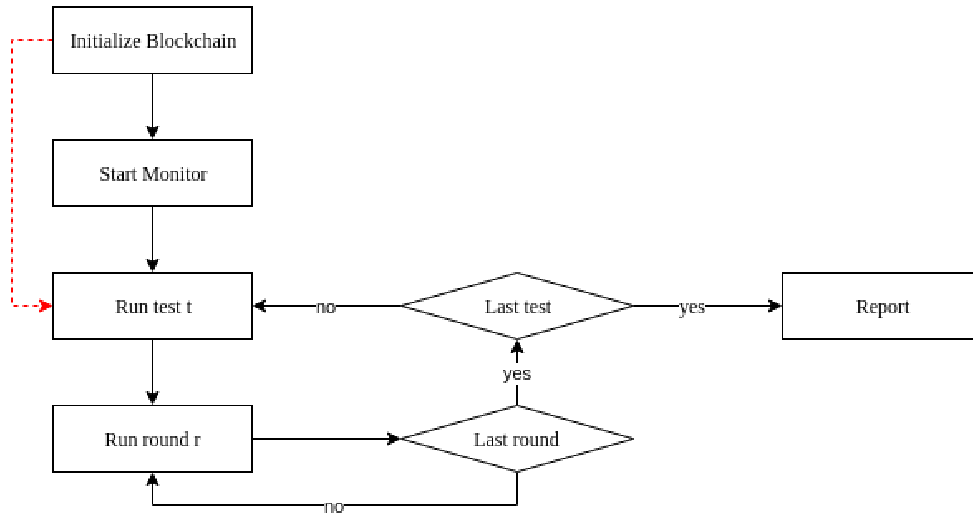

Figure 3.2: Caliper's flow diagram

Extension of Caliper

Caliper makes it possible to implement adapters as modules to benchmark other distributed ledgers. However Caliper is too over-constrained: it requires the use of their benchmarking monitor, designed for Hyperledger technologies. It assumes that other distributed ledgers have well defined API, accessible in Javascript. Some DLT such as Corda requires their own development environment (gradle), and cannot make use of the monitor because not all required parameters are given: such as a method that gets the transaction's state.

The red dashed line on Figure 3.2 bypasses the monitor to benchmark other DLT solutions directly from the DLT's client. The following sections partially show the modules' integration into Caliper. Furthermore, as Caliper is not stable yet and changes its structure quite often, the benchmarks are mostly created directly from the module root directory and not from the Caliper root directory. The following emphasizes the main steps as end-points of the module and omits their integration with the caliper root. The integration into Caliper requires modifications of the caliper-core module and a few manipulations not so relevant for the thesis.

3.3.2 BFTSMaRt

Overview

BFTSMaRt is not a DLT, but can be tested exactly in the same way. The difference here is that there is no overhead in embedding transaction in smart contracts and instantiate lots of data structures. In fact, BFTSMaRt is only the top layer where consensus on replication occurs.

The following emphasizes: how to deploy BFTSMaRt and how performance indicator data is generated on a cluster.

Deployment on Cluster

The current goal is to deploy the file system from Section 2.2 on all servers (cheetahs and cougars).

The first step is to clone the github repository or just copy the module:

https://github.com/jutinguely/caliper/packages/caliper-bftsmart

or scp[8] the files manually onto *worker@trinity*, the managing server.

All well implemented clusters should have an authority that manages the cluster's servers in order to simplify their execution. Trinity offers commands that distribute tasks to servers, two main commands are:

- trexec *-SCRIPT -SERVERS*, to execute commands on all servers, and
- trcp *-FILE -SERVERS*, to copy files on all servers.

Deploying nodes is done once from the *$HOME/caliper/packages/caliper-bftsmart/cluster*, with the bash script:

setup_trinity_bftsmart.sh -PATH -MACHINES

For example, to deploy it on 4 servers: *./setup_trinity_bftsmart.sh /home/-worker/julien cougar1-4*.[9]

Benchmark Generation

In this subsection, the client implementation and the benchmark deployment are presented. Note that the library is assumed to be deployed across servers.

Please look at the two files in the github repository: *ThroughputLatency-Client.java* and *ThroughputLatencyServer.java* for concrete implementations. These

[8] secure copy protocol via SSH
[9] see code A.1.1

two files adhere to the generic view of a client and a server from Section 3.3.1. These files give a basis for other benchmark applications.

If you decide to modify the class files or configurations files, you must propagate the modifications to all servers: it can be done by calling the following command again or a modified version of it from the manager:

./setup_ trinity_ bftsmart.sh -PATH -MACHINES .

Once the client and server classes are ready, you need to configure the *hosts.config* and *system.config* files with the right IP and parameters (signatures, DOS, Byzantine nodes, ...).

You can modify the benchmark parameters in *launchbenchmark.sh* (request's sizes, number of threads, application to run...) and start the benchmark from the managing server (worker@trinity). The servers names and IPs must have consistently configured between the file A.1.2 and the *hosts.config* configuration file.

For example to launch the increment benchmark run from the /cluster subfolder:

./launchbenchmark.sh -NUMBER_ TRANSACTIONS.

The results are stored in JSON format and split in: input parameters, output parameters and indicates the metrics used. This format is then easy to manipulate. You can find them in the cluster's shared folder */nas/scratch/julien/bftsmart/results*, or a similar shared folder that you specified in *launch_ benchmark.sh*. Each file name contains the number of threads, the request size and the thread ID that generated it.

3.3.3 Corda

Overview

Corda's trivial way to benchmark Cordapps is via JUnit. From there it is quite easy to get an idea on how long processes or methods take to execute. However, we want to test deployed networks.

Challenges highlighted here are: deploying network in an efficient way, implementing a notary cluster, connecting a Corda's client to the network, and connecting the client to Caliper.

Deployment on Cluster

Corda has a module to generate the nodes' filesystem. The user can choose between Cordform or Dockerform.[10]

The choice was to use Docker to facilitate the network's deployment. However, the use of Dockerform requires some modifications in order to make it work. The following script, located in *caliper-corda/cordapps*, modifies the current node structure to make deployment with docker-compose possible:

./configure_for_ docker.sh -PROJECT_ NAME[11]

It takes about a minute to compile the project, create the applications and corda's jars to create corda's nodes structure (structure from 2.3), located in */build* cordapp's subfolder, and build the containers (PartyA, PartyB, PartyC and Notary).

Once this bash script finishes, you find Corda Nodes embedded in Docker Containers. You can either 1) start them locally and log in, or 2) deploy them on a cluster:

1. You can simply go into */cordapps/PROJECT/build/nodes* and execute *docker-compose up* to start the network. It takes about a minute and the nodes can be accessed only once that all ports have been opened. Then open another shell and connect to PartyA with the command *ssh localhost -p 2223 -l user1* and the pre-defined password *test.* From there, you can see the flows by typing *flow list*, see its peers (*run networkMapSnapshot*) or start flows.

2. If you modify a node's file, you can just regenerate the container with *docker-compose up –build –d.* Docker takes quite a lot of memory (300MB per peer), use *docker system prune* to remove unused containers.

[10] https://docs.corda.net/head/generating-a-node.html
[11] command requiring material from sub-folder cordapps/utils

When the container images exist for each party (look with *docker ps*), you can export the images into zip files with *Docker export notary > notary.gz* for each container name (parties). The containers are zipped and placed into the folder where the operation is done (most probably under PROJECT/build/nodes). If working locally, export it on the cluster with *scp ./containers/* worker@trinity:/nas/scratch/PATH/containers*. Then use *trcp* to export the zip onto other servers and then *docker import notary.gz notary* to import them on the server's docker instance.

Lastly, you need to setup a Docker Swarm network between peers to enable communication. Choose a server to be the leader, log in via *ssh cheetah1* and type *Docker swarm init*. You get a token back. Other servers (hosting other parties) can join the Swarm with *docker swarm join -TOKEN*.

From there, the required servers are connected over an overlay network. They need to change their *docker-compose.yml* file such that all peers can join the network. Please look at Figure 3.3 to fill in your file. The use of *docker stack deploy* is also possible but not developed here.

```
version: '3'
services:
        notary:
                container_name: notary
                command: ./run-corda.sh
                image: notary
                ports:
                        - 7051:7051
                        - 8051:8051
                        - 9051:9051
                        - 2222:2222
                networks:
                        - webnet
         partya:
                 container_name: partya
                 image: partya
                 command: ./run-corda.sh
                 ports:
                         - 7052:7052
                         - 8052:8052
                         - 9052:9052
                         - 2223:2223
                 networks:
                         - webnet

networks:
        webnet:
                driver: overlay
```

Figure 3.3: Corda's Swarm configuration

Notary Cluster

Notary Cluster is Corda's core concept. However, its actual deployment and implementation is badly documented. It does not seem to be fully ready yet. Furthermore this central concept is unfortunately not implemented in its OS

version and requires the Enterprise paid version.

For these reasons, in the thesis, only one validating ordering node is used.

Corda's Client

There are three ways to connect to Corda nodes: either through their shells with ssh, through a rpc client, or through rpc via a webserver.

The first has previously been achieved with the caliper client to automate the process of sending transactions at a high rate. The performance was astonishingly low due to the ssh connection overhead.

To bypass this and get better throughput, the use of Corda's native environment is necessary (via rpc). That is the main reason why this package does not use the Caliper client and monitor frameworks (see Figure 3.4). The caliper-corda module:

- deploys the nodes as docker containers,
- starts the webserver which is bound to its Cordapps,
- sends http-request to request benchmarks, measure performances in JVM, and
- logs JSON results containing all measurements into a file.

Benchmark generation

In order to launch a benchmark you will first need to have a Cordapp to benchmark, either by implementing from scratch from the Cordapp-template or by taking an example from *caliper-corda/cordapps/**. You will then execute:

./configure_for_docker -PROJECT_NAME

and start the containers with *docker-compose up.*

Once done, go into /PROJECT/ and run

./gradlew runTemplateClient

to start the *Client.java* or *Client.kt* class from the Client module (either from intellij or from a terminal). They have to follow the same structure as in Section 3.3.1 (refer to the projects implemented in the thesis).

All parameters are defined directly from within the class itself. From there, the results are given at the end of the benchmark and written in the cordapp's sub-folder results.

Another way to automate the whole process is to use the web-server as shown on Figure 3.4. This is explained in the README.md file in the caliper-corda directory. This is implemented in the *caliper-corda/lib* sub-directory. The code is self-explanatory.

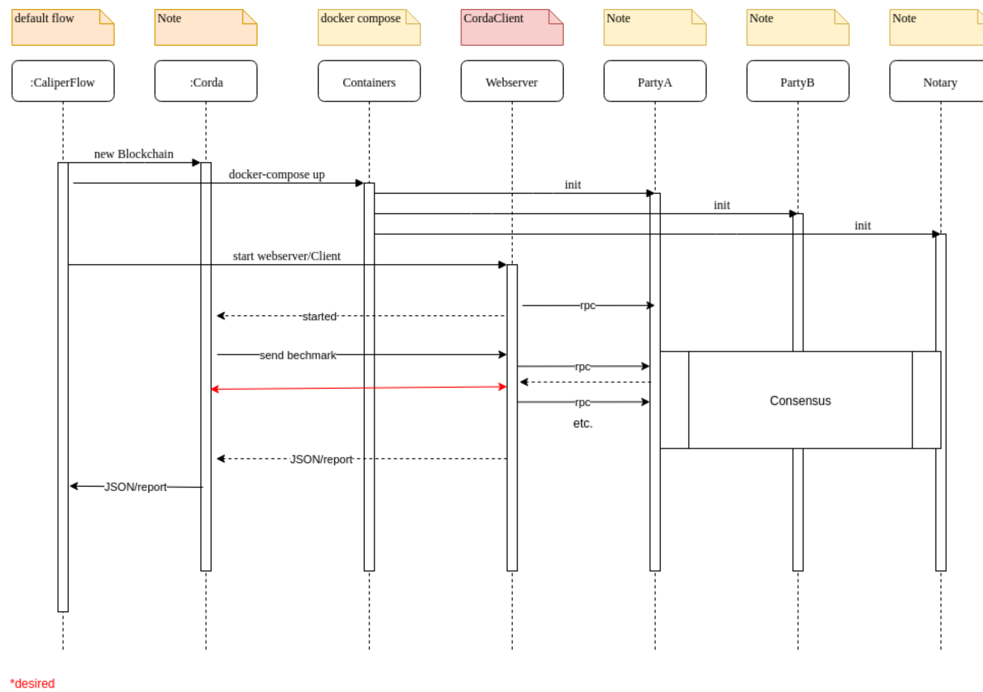

Figure 3.4: caliper-corda flow diagram

Figure 3.4 shows the problem when trying to integrate in *caliper-core*. The use of the monitor is not possible and has to be bypassed by a Client's webserver listening to Caliper's requests[12].

[12] read more in README.md

3.3.4 Fabric

Overview

As both Caliper and Fabric are Hyperledger projects, getting benchmark locally in Fabric is simple: just use caliper. The steps are self-explainatory and therefore not detailed in the report.

Deploying them on a cluster is another story, although Caliper proposes a way using Zookeeper[13]. This section describes an alternative deployment method with Docker Swarm and Docker Stack. It will then give a summary on benchmarking it on a cluster.

Deployment on Cluster

Here, a way to deploy Fabric on a cluster is presented. The explanations refer to the Trinity cluster and thus might differ from the reader's cluster. The code can be found under *caliper-fabric/cluster* It assumes all requirements from Section 2.4 have been installed on all servers. The following concern: 1) defining a network setup, 2) installing mongoDB in multihosts, 3) binding servers in a Docker Swarm and Docker Stack, 4) starting Fabric's components on all servers, and finally 5) running a chaincode, all on the cluster.

1. All deployed networks are composed of servers, each one hosting an organization. Each organization contains at least: two peers (one endorser) and a certificate authority. All can contain an orderer. In our increment use case only one Ordering service is implemented as "Solo" on cheetah3, representing the network administrator. cheetah5 and cheetah6 hold two organizations (seen as the client-supplier relation). The reader can open new terminals for different ssh sessions to the servers.

2. Every server instance needs to have a mongoDB instance running. The reader can install it with *sudo apt-get install mongodb*. The ports and connection parameters need to be modified by executing:

   mongoose.connect('mongodb://cheetah3:27017,...,cheetah5:27017/admin?replicaSet=rs0')

   in the files database.js. Once installed, the script *./startDB.sh* needs to be called on each server. The leading server will need to open a new terminal and run *mongo*.

3. Like in Corda, servers are communicating through a Docker Swarm: initiate a Swarm Orchestrator with *docker swarm init*, make others join with

[13] ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

docker swarm join-token manager, join other nodes to the swarm network as "managers" with *docker swarm join –token TOKEN* and finally create an overlay network with *docker network create –attachable –driver overlay fabric*.

4. Once steps 1-3 are done, docker containers can be started. For that, the reader opens a new terminal for each server and runs the script *./startCluster.sh*. You can verify that your network is up and running thanks to *docker ps, docker service ls, docker node ls, and docker stack ls*. The file is self explanatory.

5. You can then launch your Chaincode either with Fabric's SDKs or manually from a terminal by following this tutorial:

   https://hyperledger-fabric.readthedocs.io/en/stable/install_instantiate.html

There are other ways to create a Fabric network, Caliper uses for instance Zookeeper instead of Docker Swarm. Both make sense if they run the Fabric's kafka consensus algorithm. This tutorial[14] explains how to make use of it.

Benchmark Generation

As the client's implementation, using Docker Swarm/Stack is not finished, the benchmarks are generated with the help of Caliper. In order to deploy your network, you can use Zookeeper for now:

1. Start Caliper's zooservice: *caliper zooservice start*

2. Launch a caliper-zoo-client on eah target machine using Caliper CLI: *caliper zooclient start -w /myCaliperProject -a <host-address>:<port> -n my-sut-config.yaml*

3. Modify the client type setting in configuration file to "zookeeper":

   For example:

```
"clients": {
   "type": "zookeeper",
   "zoo" : {
     "server": "10.229.42.159:2181",
     "clientsPerHost": 5
   }
 }
```

4. Launch benchmark as explained on caliper on any machine.

[14] https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html

Chapter 4

Measurements

In this chapter, the results from the benchmarks are presented, explained and interpreted. A simple application is mapped to each DLT solution in order to compare their performances (see code A.2 for the increment example). The section with an asterisk are here to add more examples, but do not help in comparing DLTs.

While reading the results, keep in mind the throughput of some interesting distributed systems as a basis for comparison:

VISANet:	$\sim 20'000 <$ $[tps]$
Bitcoin:	~ 6 $[tps]$
Ethereum:	~ 15 $[tps]$

Table 4.1: VisaNet, Bitcoin and Ethereum throughput

4.1 Results

4.1.1 BFTSMaRt

Overview

In the first benchmark session, we are interested in looking how this replication framework 1) scales with many servers involved in the consensus protocol, 2) impacts multi-threading, 3) the response size and 4) signatures have on throughput. Therefore, we run different benchmarks in function of:

1. the number of servers (2, 4, 8, 16),
2. the number of threads (1, 10, 50, 100),
3. the request/response size (4, 64, 512, 1024 [bytes]), and

4. signed and not signed transactions (signed, not signed),

using the increment example from annex A.2.

We are interested in looking for the optimum number of threads. Having too many threads deteriorates performance (overhead in synchronization) and too few might not use the computer's ability to process transactions in parallel: the client needs to wait on the previous transaction to go on with the next[1].

In the following pages the reader will find the results of the 128 benchmarks, each one running 1′000′000 transactions. The standard deviation is calculated based on a 68.27% confidence interval ($[\mu - \sigma, \mu + \sigma]$). The execution took about 18 hours 43 minutes.[2]

Figure 4.1 shows: the connection from a remote desktop to the cluster manager. From the manager all script are launched to deploy the BFTSMaRt library and commands to a subset of servers or to clients.

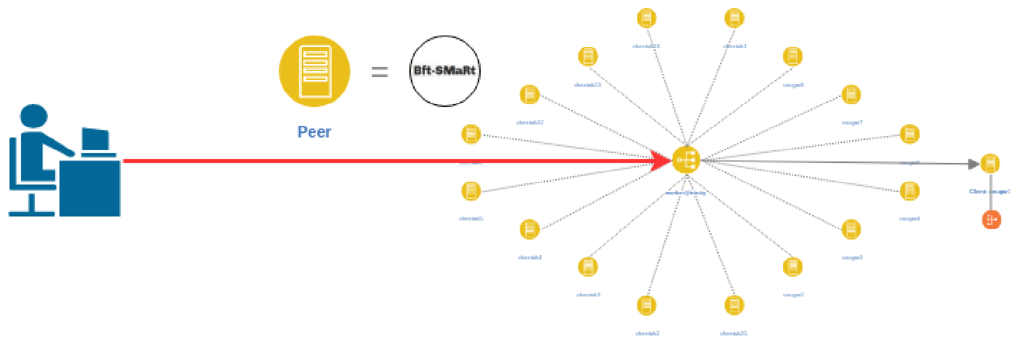

Figure 4.1: BFTSMaRt: benchmarking setup

[1] To know more about pipelining, please read https://en.wikipedia.org/wiki/Instruction_pipelining.
[2] It takes into account the setup time.

No Signature

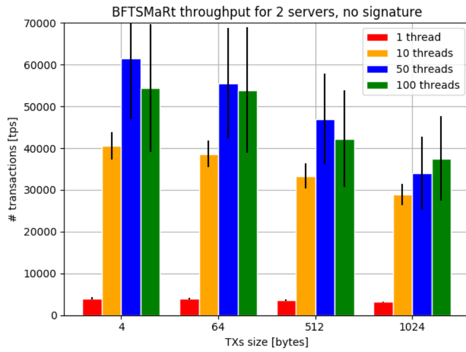

Figure 4.2: BFTSMaRt: 2 servers, no signature

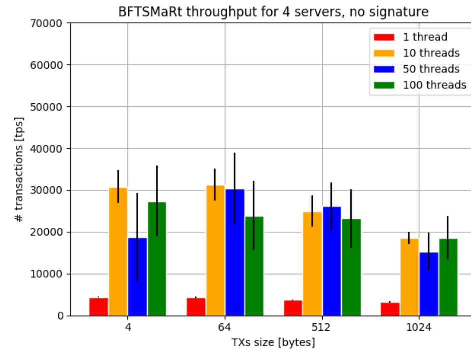

Figure 4.3: BFTSMaRt: 4 servers, no signature

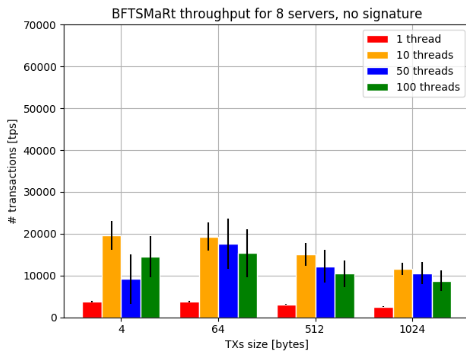

Figure 4.4: BFTSMaRt: 8 servers, no signature

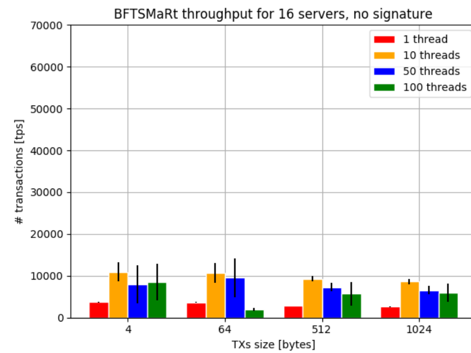

Figure 4.5: BFTSMaRt: 16 servers, no signature

Multiplying the number of servers by 2 seems to reduce the throughput by a factor of about 2. This can be shown from Figures 4.2 to 4.5. Therefore, the main overhead with no signature lies in the number of servers participating in the network.

The optimum number of threads for this application varies, although 50 is a good candidate for low number of servers and 10 for higher number of servers. This number decreases with more servers. Request sizes impact the throughput in a difference of few 1000s transactions per second between the different sizes.

With Signatures (change of scale!)

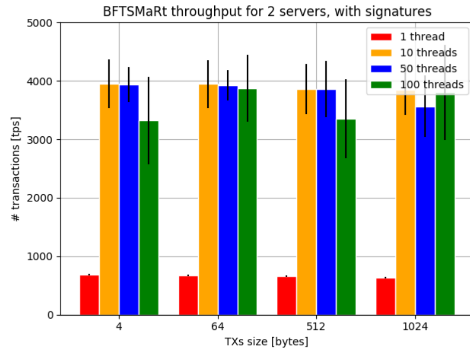

Figure 4.6: BFTSMaRt: 2 servers, with signatures

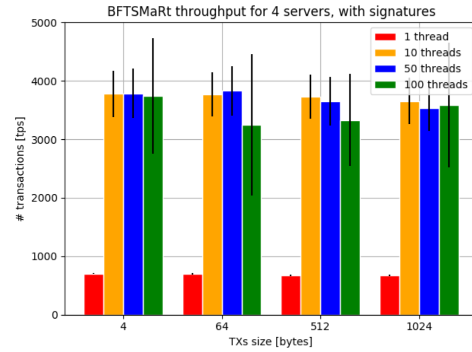

Figure 4.7: BFTSMaRt: 4 servers, with signatures

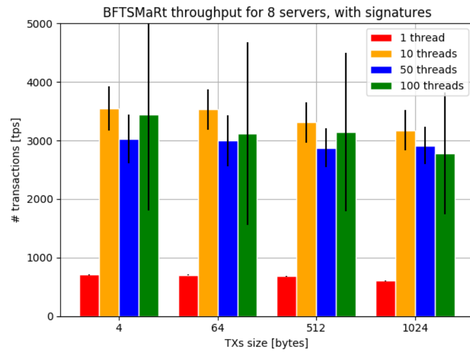

Figure 4.8: BFTSMaRt: 8 servers, with signatures

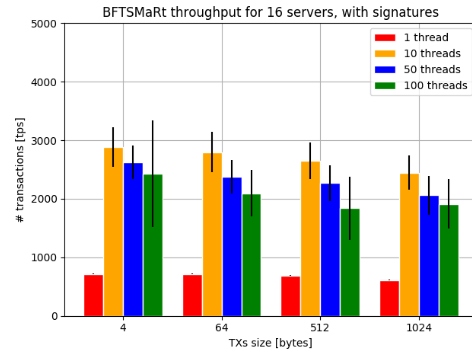

Figure 4.9: BFTSMaRt: 16 servers, with signatures

The number of servers involved in the consensus does not affect the throughput as much as with no signature.

From 10 threads adding threads does not affect the performance that much. Therefore, the optimum seems to be 10 threads. (see Figure 4.9)

Request sizes do not affect throughput as much as with no signature (they have almost the same shape).

Interpretation

BFTSMaRt is really optimized. Adding more servers in the consensus does not add a huge overhead in the system. Communication is of course more important with many machines.

The request/response sizes do not play a role when transactions are signed. Therefore, the main overhead of BFTSMaRt probably lies in signatures, which cost a lot and reduce the throughput by a factor between 3 and 10.

In general, having a high request/response size impacts the throughput as shown in the graphics. Interestingly, more threads does not mean more throughput; such benchmarks can help finding the right number for a given application.

A high standard deviation is observed in almost all measurements. This comes from the addition of all communication waiting time in queues, which varies.

The benchmarks with signatures would have probably been more interesting with higher values (sizes, number of threads), as signatures are the throughput's overhead here.

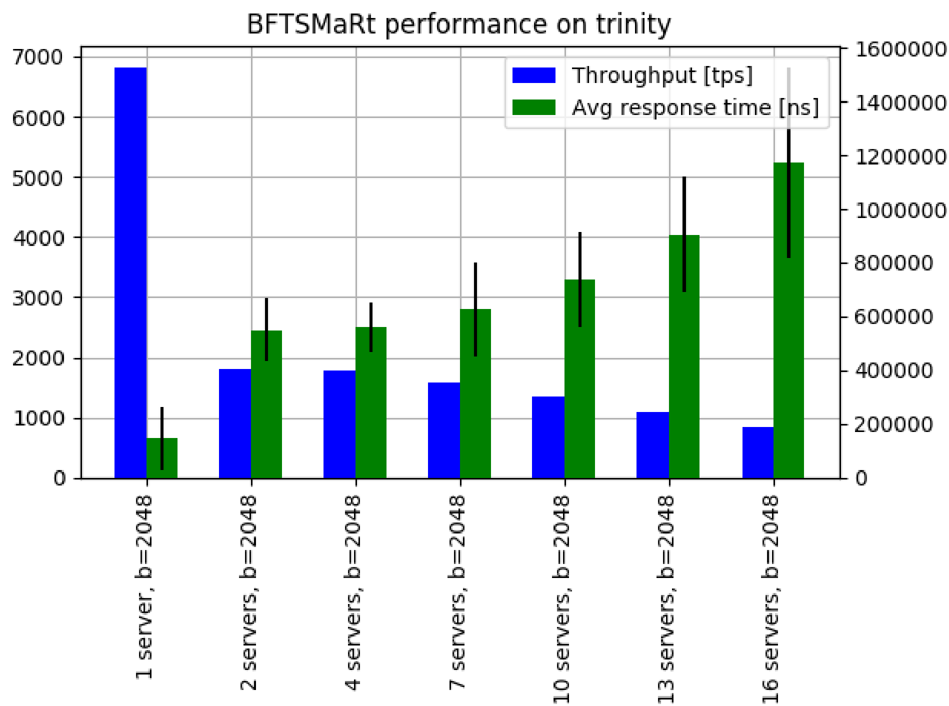

Figure 4.10: BFTSMaRt: effects due to number of servers

Figure 4.10 shows the impact of the number of servers running the increment example for a request size of 4 bytes (enough to hold an integer). The reader can

see that when the server just writes on one server, it goes 4 times faster, because it does not need to communicate. The number of server n is chosen, allowing f byzantine nodes and satisfying the equation: $3f + 1 \leq n$.

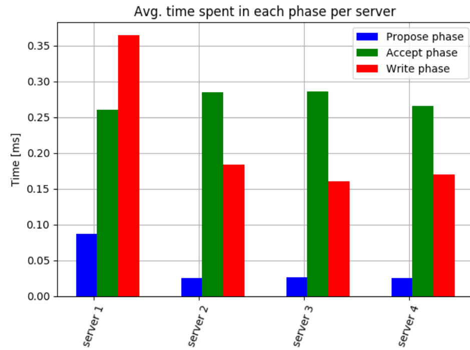

Figure 4.11: BFTSMaRt: time per phase

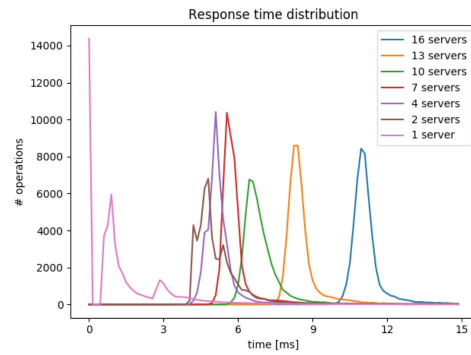

Figure 4.12: BFTSMaRt: response time distribution

Figure 4.11 shows the average amount of time spent in the phases: PROPOSE, ACCEPT and WRITE with 4 servers executing the increment example. In this benchmark, we see that server 1 is the leader because times within the PROPOSE and WRITE phases are higher. The ACCEPT phase duration is about the same for all.

Figure 4.12 shows the response times' distributions for each benchmark, running the increment example.

4.1.2 Corda

Overview

This benchmark session exposes the results of the increment example for Corda. It then presents the performance results of a financial real use case in Corda in the second part to give a real use case benchmark.

We want to measure the throughput for the 4 different examples and study the effects of[3]:

- the number of threads (1, 10, 50, 100, 500, 1000)[4], and
- Corda OS[5] and Corda Enterprise[6]

on throughput. For this, we use 10′000 transactions, two parties and one notary[7]. One benchmark takes between 20 and 50 minutes. The standard deviation is calculated based on a 68.27% confidence interval ($[\mu - \sigma, \mu + \sigma]$).

As shown on Figure 4.13, the number 1 represents a notary validating an update to the ledger (hence an increment operation or an issuance). The number 2 represents a notary validating a transaction between peer1 and peer2. In the second part, number 1 is used to issue cash, number 2 is used for the payment example.

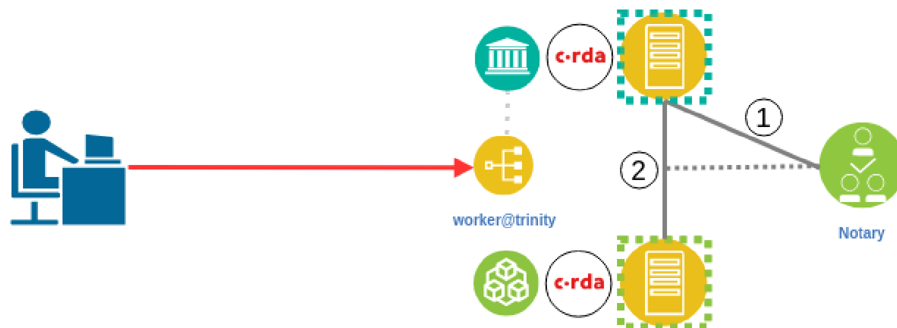

Figure 4.13: Corda: benchmarking setup

[3]Here we do not analyze the effect of request/response size as it would require to change the data-structure before each deployment (hard at runtime).

[4]for the same purpose than in BFTSMaRt

[5]Open Source

[6]To run Corda Enterprise, replace corda.jar in nodes directory by the Entreprise corda.jar (can be found in trial version)

[7]Notary cluster is not implemented in OS version yet.

Increment Example

The increment example had to be transcript into a Cordapp.

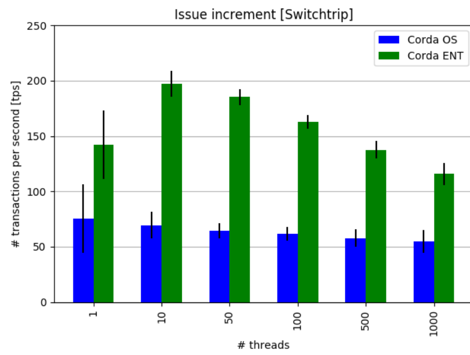

Figure 4.14: Corda: increment issue

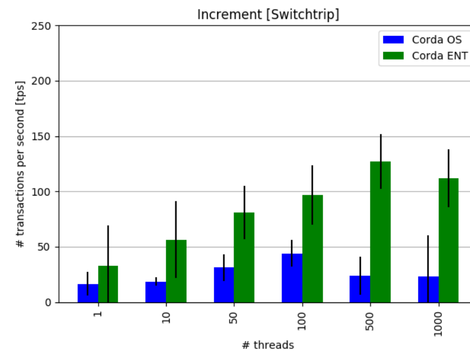

Figure 4.15: Corda: increment

Here, we see that consuming states to create a new one costs about 3 times more than just issuing transactions (see Figures 4.14 - 4.15).

* Finance Example from Corda

This example does not bring anything to the comparison of other DLTs, it just analyzes two flows from the official finance Cordapp, that could be real deployed application. "Cash Issue" aims to create accounts; it assigns an amount to an owner via a notary. "Payment" consumes an account balances and create two output accounts one for the debtor and one for the creditor. As the debtor and creditor are always identical, this application is just consuming the debtor account.

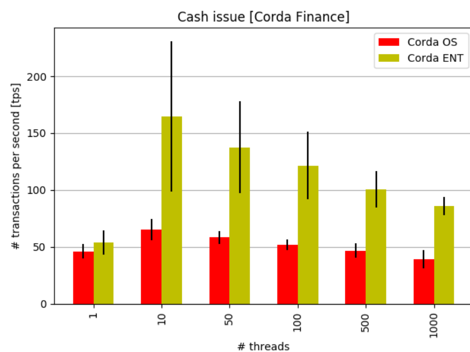

Figure 4.16: Corda: cash issue

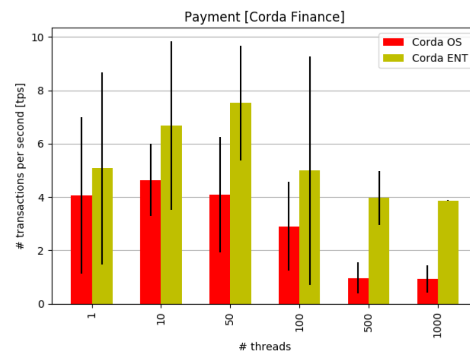

Figure 4.17: Corda: payment

In this example, the difference between creating and consuming a state is even more distinguishable. Each benchmark sent 10'000 transactions. This shows how costly it is to walk the chain to the original transaction in Corda.

Interpretation

Corda's results might seem poor. However it is necessary to look at Corda's functionalities. Corda:

1. offers a straightforward experience: it uses quite easy concepts,
2. offers good privacy because it uses point-to-point messaging, and
3. deploys Cordapps with less efforts (using high level programming languages).

More astonishing is their wish to separate multi-threading and single-threaded applications. If reasonable performance or building a notary cluster are wanted, then the developer needs to choose the Enterprise version.

These results are satisfactory for low throughput applications. The reader might want to reconsider their choice if he needs to request over hundreds transactions per second.

The results match with the ones obtained by Corda in another study [17]. It was hard to obtain the same performance, as sharding and Enterprise's source code was necessary.

Moreover, batching transactions together is not offered in Corda in contrast to Fabric. This would certainly improve performance, but contradict Corda's design at the same time.

Corda uses direct communication and does not have have to account for the time it takes to broadcast a message to the entire network. Individual transactions are sent directly to the involved parties. It is a very quick process. In our case, though, batching would have helped as we always involve the same peers.

4.1.3 Fabric

Overview

We are running the increment example on Fabric. The channel is configured with a batch timeout of 250 [ms], a maximal batch size of 100. These value can be modified on channel's creation. These values will stay constant through the benchmarks.

We use 10′000 transactions and analyze the performance of the increment application under different sending rates and output the throughput and average response time. We look at three different operations on the peer's ledger: issue, increment, query and transfer.

Figure 4.18 shows the two configurations we use. The number 1 is the configuration where we have only one peer under one organization. This case will enable comparison with Corda. The second one is useful to see how a second organization within the channel. The increment example is implemented in a chaincode[8].

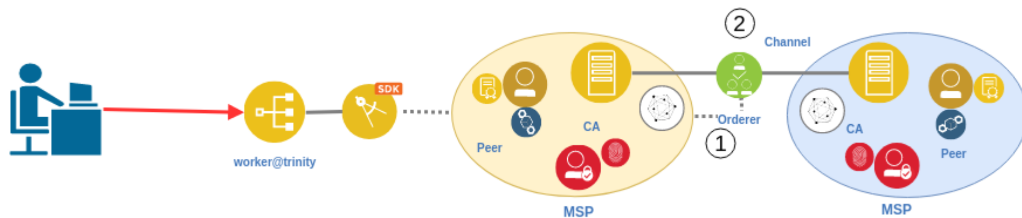

Figure 4.18: Fabric: benchmarking setup

[8]can be found on repository under benchmark/simple

Increment Example on 1 organization

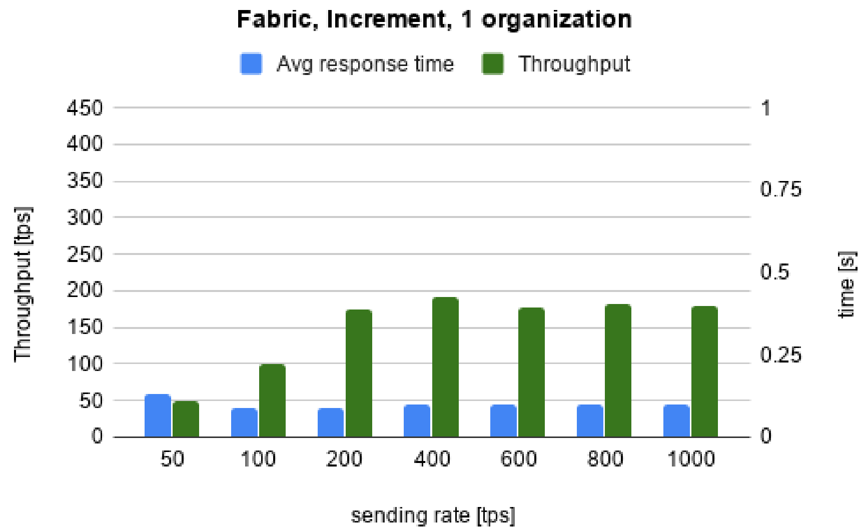

Figure 4.19: Fabric: increment on 1 organization

The throughput for incrementing a counter is 50 [tps] better than the one obtained by Corda. Augmenting the sending rate has no impact as it is bound by the network capacity. Here, we see that Fabric batches lots of transactions together as the average response time is about 0.4 [s] (about 100).

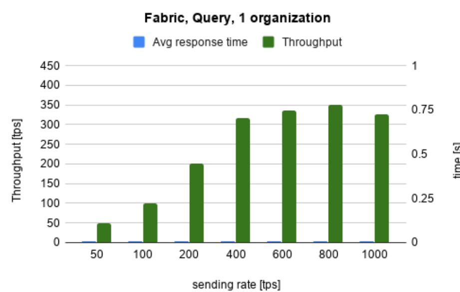

Figure 4.20: Fabric: query on 1 organization

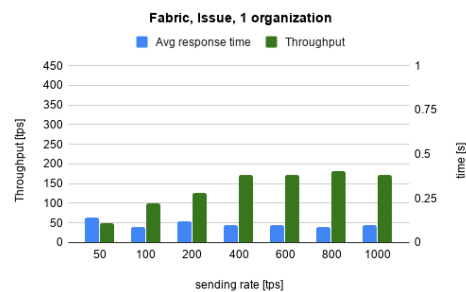

Figure 4.21: Fabric: issue on 1 organization

As in Corda's Enterprise edition, the throughput to issue transactions is bound to about 170 [tps].

* Increment Example on 2 organizations

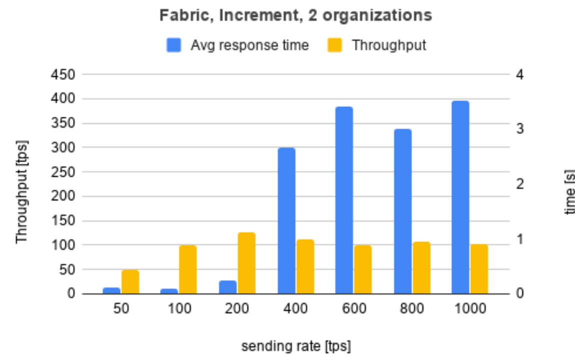

Figure 4.22: Fabric: increment on 2 organizations

Adding another organization in the Channel is not free, even with a Solo ordering service. Both peers needs to synchronize. The throughput is therefore a bit lower.

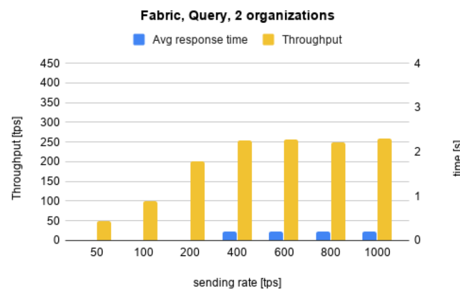

Figure 4.23: Fabric: query on 2 organizations

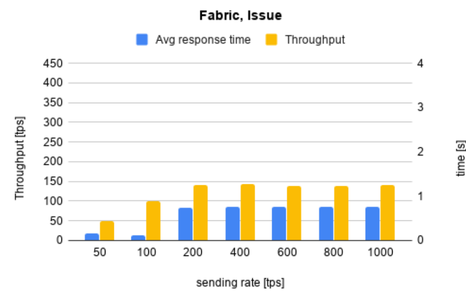

Figure 4.24: Fabric: issue on 2 organizations

Interpretation

For the increment example, both Corda and Fabric solution seem to be have similar throughput's ranges. We can expect better from Fabric with higher maximal batch sizes and configurations. This is however not shown here. Some more interesting results about Fabric's performance can be found here [12].

Chapter 5
Conclusion

This work gives insights into benchmarking DLT. On the one hand, BFTSMaRt gave high throughput due to its simple consensus framework. On the other hand, Corda achieved an astonishingly low throughput, but gives us the whole infrastructure to implement a one-to-end smart contract application. Fabric seems to be the good intermediary: a solution with reasonable throughput and functionality to implement smart contracts in a more flexible way. A downside of Fabric is its deployment complexity.

The main limitation of the work accomplished is the consensus protocols in DLTs. Our benchmarks are not taking into account this part of the transaction. Therefore throughput with the consensus part can be expected to be even lower than the results obtained. This work has defined DLT performance by the throughput. This assumption is also a limitation of the work accomplished. These systems needs to take much more into account to be evaluated and thus compared, such as: cryptography, CPU consumption, consensus algorithm used, etc.

Telling which one would be more suited for a particular application would be too subjective for now. The advise to take away from this thesis is to look at the level of throughput the application needs in order to work well. If one wants to replicate data from sensors collecting thousands data measurement per second, BFTSMaRt would be more suitable. If one wants to implement a low throughput application, such as an application that records houses' ownership, then a framework such as Corda would be more appropriate. For common use cases Fabric would probably be the best solution, modifiable enough to respond to precise needs.

These benchmarks are in some ways interesting to compare with the "marketing benchmarks" that promise high throughput. It shows how important performance is for them: giving high numbers is much more appealing than listing advantages of their solution.

Benchmarking DLT is not a trivial task. Caliper looks for developers to extend the tool, make it more reliable, and enable new DLT and parameters.

As many applications follow from DLT, many particular use cases and smart contract applications can be tested on different distributed ledgers.

Caliper or similar projects will integrate great extensions in the next few years that will enable it to benchmark any DLT. From that point on, we will be able to segment DLTs into specializations. Once that is possible, the deployment of such technology will be even easier and enable large scale deployment of smart contract applications.

Bibliography

[1] A. report by the UK Government Chief Scientific Adviser, "Distributed ledger technology: beyond block chain," *Gouvernment office for science*, 2016.

[2] J. Sousa and A. Bessani, "State machine replication for the masses with bft-smart," *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (pp. 355-362). IEEE*, Jun. 2014.

[3] S. Loignon, *Big Bang Blockchain: La seconde révolution d'internet.* Tallandier, May 2013.

[4] B. Burns, *Designing Distributed Systems.* O'Reilly, May 2018.

[5] N. El Ioini and C. Pahl, *A review of distributed ledger technologies.* Springer, Cham, 2018.

[6] C. Lima, "Developing open and interoperable dlt / blockchain standards," *Computer, 51(11), 106-111*, 2018.

[7] J. Sousa and A. Bessani, "From byzantine consensus to bft state machine replication: A latency-optimal transformation," *2012 Ninth European Dependable Computing Conference (pp. 37-48). IEEE*, May 2012.

[8] I. G. M. H. Richard Gendal Brown, James Carlyle, "Corda: An introduction," *R3 CEV, August, 2016, vol. 1, p. 15*, Aug. 2016.

[9] M. Hearn, "Corda: A distributed ledger," *Corda Technical White Paper, 2016*, 2016.

[10] E. Androulaki, "Hyperledger fabric: A distributed operating system for permissioned blockchains," *Proceedings of the Thirteenth EuroSys Conference (p. 30). ACM*, Apr. 2018.

[11] C. Cachin, "Architecture of the hyperledger blockchain fabric," *Workshop on distributed cryptocurrencies and consensus ledgers (Vol. 310, p. 4)*, Jul. 2016.

[12] S. N. Thakkar, Parth and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018*, 2018.

[13] "Throughput — a corda story," https://medium.com/corda/throughput-a-corda-story-1bc2cb9b2b60, accessed: 2019-09-27.

[14] S. C. . T. S. Pongnumkul, S., "Performance analysis of private blockchain platforms in varying workloads," *26th International Conference on Computer Communication and Networks (ICCCN) (pp. 1-6). IEEE*, Jul. 2017.

[15] e. a. Gorenflo, Christian, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second." *arXiv preprint arXiv:1901.00910 (2019)*, 2019.

[16] "Caliper," https://hyperledger.github.io/caliper/, accessed: 2019-08-13.

[17] "Sizing and performance," https://docs.corda.r3.com/sizing-and-performance.html, accessed: 2019-09-23.

Appendix A

Annexes

A.1 BFTSMaRt Benchmarks bash script

A.1.1 setup_trinity_bftsmart.sh

```
#!/bin/bash

PATH=$1
MACHINES=$2

trexec "rm -r $PATH/library-1.1-beta" $MACHINES

trcp library-1.1-beta/ $MACHINES

trexec "mv ./../library-1.1-beta/ $PATH" $MACHINES
```

A.1.2 launchbenchmarks.sh

```
#!/bin/bash
# @Julien Tinguely 2019

# 1. configurations: lauch number of nodes 1 2 4 8 16 --------------------------
# - modify hosts.config, system.config IPs
# - set maxbatchsize, #faulty?

# arg number of transactions to send
transactions=$1
# machine cougar6 is the client
client=cougar6
# output file base name
dir=results_BFTSMaRt_${transactions}

# create folder for results for each setup (2,5,8,16)
echo "Create directory for results"
trexec "mkdir /nas/scratch/julien/$dir/" $client
trexec "mkdir /nas/scratch/julien/$dir/2servers" $client
trexec "mkdir /nas/scratch/julien/$dir/4servers" $client
trexec "mkdir /nas/scratch/julien/$dir/8servers" $client
trexec "mkdir /nas/scratch/julien/$dir/16servers" $client

# assign machines used for benchmarks
echo "Start session"
servers2=(cougar1 cougar2)
servers4=(cougar1 cougar2 cougar3 cougar4)
servers8=(cougar1 cougar2 cougar3 cougar4 cougar7 cougar8 cheetah1 cheetah2)
servers12=(cougar1 cougar2 cougar3 cougar4 cougar7 cougar8 cheetah1 cheetah2
 cheetah3 cheetah4 cheetah5 cheetah6)
servers16=(cougar1 cougar2 cougar3 cougar4 cougar7 cougar8 cheetah1 cheetah2
 cheetah3 cheetah4 cheetah5 cheetah26 cheetah22 cheetah23 cheetah24 cheetah25)

# Each machine has a corresponding id (see config/hosts.config -/system.config)
declare -A ids=( ["cougar1"]=1001 ["cougar2"]=1002 ["cougar3"]=1003
 ["cougar4"]=1004 ["cougar7"]=1005 ["cougar8"]=1006 ["cheetah1"]=1007
 ["cheetah2"]=1008 ["cheetah3"]=1009 ["cheetah4"]=1010 ["cheetah5"]=1011
```

```
 ["cheetah26"]=1012 ["cheetah22"]=1013 ["cheetah23"]=1014 ["cheetah24"]=1015
 ["cheetah25"]=1016)

# 2. For arg setup do benchmarks request_size x #cli --------------------------
launch () {
  servers=($@)
  client=cougar6
  len=${#servers[@]}
  folder="$len"servers
  # print output file name
  echo "name of folder: results_BFTSMaRt/$folder"
  # iterate on resquest size
  for size in 4 256 1024
  do
    # iterate on number of threads
    for cli in 1 10 50 100
    do
      echo "A. Benchmark on $len servers with $cli clients and requests size
       $size bytes -----"

      # start all servers
      echo "B. BRING all servers up"
      for s in ${servers[@]}
      do
        echo "$s with id ${ids[$s]}"
        trexec "cd /home/worker/julien/library-1.1-beta/;
        ./runscripts/smartrun.sh bftsmart.demo.benchmark.ThroughputLatencyServer
                ${ids[$s]} $size false 0 0" $s;
        echo "Hello"
        sleep 2
      done

      echo "B2. Waiting 6 sec"
      sleep 6

      # start the client
      echo "C. Starting benchmark on $len servers with $cli clients
      , requests size $size bytes and $(($transactions/$cli)) txs"

      trexec "cd /home/worker/julien/library-1.1-beta/;
  ./runscripts/smartrun.sh bftsmart.demo.benchmark.ThroughputLatencyClient 1001
      $cli $(($transactions/$cli)) $size 0 true 0" $client;

      #echo "Wait $(($cli*20)) sec to complete (better too long than too short!)"
      #sleep $(($cli*20))
      sleep 2

      echo "D. Move results to /nas/scratch/results"
      # dont forget to change folder name 4servers
      trexec "mv /home/worker/julien/library-1.1-beta/results/*
            /nas/scratch/julien/$dir/$folder/" $client
    done
    # kill servers all threads and processes hanging!
    echo "E. Kill servers"
    trexec "pkill -f 'java'" $client
    for s in ${servers[@]}
    do
      echo "kill java processes on ... $s"
      trexec "pkill -f 'java'" $s
    done

    echo "F. Wait 15 sec ---------------------------------------------------------"
    sleep 15

  done

}

# setup means replacing the system.config file before the benchmark launch
# config depends on the servers.
setup () {

  servers=($@)
  len=${#servers[@]}

  trexec "rm /home/worker/julien/library-1.1-beta/config/currentView" $client

  trexec "cp /nas/scratch/julien/SystemConfigs/$len/system.config
          /home/worker/julien/library-1.1-beta/config" $client

  trexec "cp /nas/scratch/julien/SystemConfigs/$len/hosts.config
          /home/worker/julien/library-1.1-beta/config" $client

  for s in ${servers[@]}
```

```
  do
    echo "setup $s"
    trexec "rm /home/worker/julien/library-1.1-beta/config/currentView" $s
    trexec "cp /nas/scratch/julien/SystemConfigs/$len/system.config
            /home/worker/julien/library-1.1-beta/config" $s
    trexec "cp /nas/scratch/julien/SystemConfigs/$len/hosts.config
            /home/worker/julien/library-1.1-beta/config" $s
  done

}

# benchmarks ----------------------------------------------------------------

setup ${servers2[@]}
launch ${servers2[@]}
sleep 10
setup ${servers4[@]}
launch ${servers4[@]}
sleep 10
setup ${servers8[@]}
launch ${servers8[@]}
sleep 10
setup ${servers16[@]}
launch ${servers16[@]}
sleep 10

#----------------------------------------------------------------------------
```

A.2 Increment example

Algorithm 3 Server

$c \leftarrow 0$
while *up* **do**
  $service.receive(increment())$
  ...consensus and replication...
  $c \leftarrow c + 1$
end while

Algorithm 4 Client, [n: Number of transactions]

1: $nTX \leftarrow n$
2: $i \leftarrow 0$
3: **while** $i \leq nTX$ **do**
4:   $service.send(increment())$
5:   record response times
6:   # wait on service's response
7:   $i \leftarrow i + 1$
8: **end while**
9: **return** results