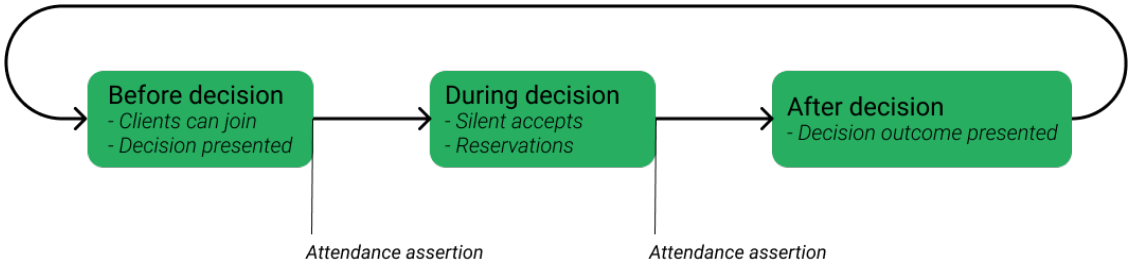


A distributed decision log for meetings within a consortium of organizations

Evaluating the feasibility of a secure, trustworthy and usable decision log for large meetings within a consortium of organizations

Master's thesis in Computer Systems and Networks

NIKLAS JONSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

A distributed decision log for meetings within a consortium of organizations

Evaluating the feasibility of a secure, trustworthy and usable decision log for large meetings within a consortium of organizations

NIKLAS JONSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

A distributed decision log for meetings within a consortium of organizations
Evaluating the feasibility of a secure, trustworthy and usable decision log for large meetings within a consortium of organizations
NIKLAS JONSSON

Supervisor: Magnus Almgren, Department of Computer Science and Engineering
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The decision process model implemented by the prototype system and evaluated (see Section 3.2).

Typeset in LaTeX
Gothenburg, Sweden 2021

A distributed decision log for meetings within a consortium of organizations
Evaluating the feasibility of a secure, trustworthy and usable decision log for large meetings within a consortium of organizations
Niklas Jonsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

When conducting meetings within a consortium of organizations, a method of recording key aspects like participation and decisions so that they can be agreed upon by all parties is needed. A decision log based on a distributed ledger; a data store that is shared on equal terms between the organizations, would remove the need to trust a central authority, and if the data structure of the ledger is a cryptographically chained log, the entries are immutable and secure against tampering. Using a Byzantine fault tolerant (BFT) consensus algorithm when reaching agreement on the state of the data store would further increase the security of the system by making it tolerant against malicious behavior from some of the organizations.

The introduction of smart contracts has broadened the field of use cases where distributed ledgers are considered. However, poor performance compared to other systems due to the high complexity of their consensus algorithms is often a limiting factor. While providing the desired functionality, the performance of a system such as this has to be sufficient to be usable; users have to see the response of their actions in near real-time and the meeting cannot be significantly delayed.

A prototype system based on these methods was designed and developed, and evaluated in terms of usability. Two metrics were defined to quantify the usability, *response time* and *overhead*. Response time was defined as the delay between a user action and its response and overhead was defined as the time added to a meeting by the system when recording the outcome of a decision. The results were measured at different scales in terms of meeting participants, network delay and number of nodes participating in the consensus method.

It was found that the performance impact of using a distributed ledger with a BFT consensus algorithm is considerable. The response time remained at sufficiently low levels, between 1.4 and 3.4 seconds. However, due to the high message complexity of BFT consensus methods, the overhead of the system was too high for the system to be considered practical; between 12 and 110 seconds for a single decision. A high number of users especially affected the overhead negatively, but the number of participating organizations and the amount of network delay also had a significant effect.

In conclusion, distributed ledgers in combination with Byzantine fault tolerance provides a high level of trust and security to a system, but performance is a big issue that limits the reasonable uses of the technologies. If a system is designed with these technologies, special care has to be taken to keep the rate of events low so that the performance remains sufficient.
Keywords: distributed systems, distributed consensus, distributed ledger technology, blockchain, Byzantine fault tolerance.

Acknowledgements

I would like to thank Yolean for providing me with an interesting topic and a lot of technical help. In particular, thanks to Staffan Olsson for his interest and expertise and Anton Lindgen for his support and much needed afterworks. I would also like to thank my supervisor Magnus Almgren for his time, patience and willingness to take this thesis on when things did not go as planned the first time around. Finally, I would like to thank my former roommate Arvid for his great company during the first few months of this thesis, and Yolean for a spacious office to go to when the pandemic gave us more company than we initially imagined.

Niklas Jonsson, Gothenburg, March 2021

Contents

List of Figures

List of Tables

1
Introduction

With increased digitalization, remote meetings become more popular and increasingly viable as an alternative to traditional face-to-face meetings. Remote meetings have several advantages; meetings can be scheduled without concern for the physical location of the attendants and reduced travel leads to time and money saved and a lower environmental impact. They also have the possibility of reducing disease transmission which is a current issue due to the coronavirus outbreak that started in late 2019.

However, there are also several problems and drawbacks with remote meetings compared to meetings in person. Technical issues can cause problems and the exchange of information might not be as clear as in meetings in person. For example, in a meeting between representatives of different organizations in a consortium in the construction industry, the different organizations might not trust each other and are not necessarily aligned in terms of economic interests or goals, so disagreements can quickly lead to large costs or lawsuits. When decisions are taken in such a context, knowing and agreeing on *who was in attendance*, *what information they had taken part of* and *whether or not they opposed a decision* becomes important.

An aid to overcoming some of these problems would be a system in which binding decisions can be taken and important information about these decisions can be recorded in a way so that it can be trusted and relied on by all parties in the consortium. This project proposes such a system, providing a shared and immutable decision log based on a distributed ledger where information can be stored in strictly defined ways agreed upon by all parties. Such a system would be able to serve as a trusted source of truth in disputes because of its immutability and strictly defined transactions and each member of the consortium would be able to own and run their own equal replica of the data to increase trust and provide protection against data tampering.

1.1 From Bitcoin to Smart Contracts

The idea of using distributed systems to securely decentralize and distribute data and operations on that data is not unique. A current example of this is blockchains and the increased use of smart contracts to solve more general problems than just cryptocurrency exchange and mining.

The concept of blockchains was popularized in 2008 by the original Bitcoin paper [21], released by an unknown person or group under the pseudonym Satoshi Nakamoto. Bitcoin is based on previous research and ideas like b-money [9], which

in 1998 proposed a protocol for money exchange and contract enforcement without the need for central authorities like government institutions, and Hashcash [2], which proposes a proof-of-work algorithm as a denial-of-service counter measure.

In short, proof-of-work based consensus method is based on guessing values that begins with a certain number zeros when hashed. Hash functions are *one-way functions*, which means that they are easy to calculate but hard to invert. In proof-of-work, this means that finding a value that, when hashed, ends with a certain number of zeros can only be done by guessing while checking that the found hash is valid is easy.

In Bitcoin, a coin is defined as a chain of digital signatures and the ledger consists of *cryptographically linked blocks of transactions*. A proof-of-work based consensus method is used to reach consensus about the state of the blockchain, while still allowing anyone to participate in the network. If a node finds a value that begins with the given number of zeros when hashed, it gets to insert the next block in the chain. All blocks contains a pointer to the previous block, so to be able to change a block, you would not only have to guess correctly once, but once for each previous block as well, making the blocks cryptographically linked. This method works well, but is extremely computationally expensive since a main premise is that it is hard to guess the correct value and a lot of invalid guesses have to be made.

Since Bitcoin was released, many competing cryptocurrencies based on similar blockchains have been developed, and some systems generalize the transactions to handle any change of state in the data. One example is Ethereum, which in its white paper describes Bitcoin more generally as a state transition system:

> "From a technical standpoint, the ledger of a cryptocurrency such as Bitcoin can be thought of as a state transition system, where there is a "state" consisting of the ownership status of all existing bitcoins and a "state transition function" that takes a state and a transaction and output a new state which is the result" [6].

The idea of Ethereum as a more general blockchain, not purpose-built for a cryptocurrency, is also described:

> "What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code" [6].

Many other blockchain systems supporting arbitrary state transitions in the form of smart contracts have been proposed and developed since Ethereum. The advantages of a blockchain system in terms of the very high level of distribution and decentralization are obvious, and many industries have realized the value it can add. There are, however, some major drawbacks that come with the level of distribution that blockchains provide, that seems to make them suitable only in a very limited set of applications. Most importantly, the consensus methods needed to ensure correctness between the replicas are often very complex and the time required to reach consensus is often very high compared to less decentralized solutions [8].

1.2 Context

This thesis was done at Yolean AB, and as a part of the research project *KIDSAM: Knowledge and information-sharing in digital collaborative projects*. Yolean AB offers software that supports lean management, mainly a visual planning application that is used by, among others, several construction companies. KIDSAM aims to create an increased understanding of how technology can support the connected industry to conduct collaborative projects where people still have a central role. Two types of collaborative projects are defined:

- *Long-term collaboration,* where the process is characterized by preventive maintenance over the course of several years;
- *Short-term collaboration,* which is characterized by a shorter and more intensive process where decisions need to be taken and where traceability, transparency and clarity regarding these decisions can be valuable throughout the lifetime of the product. [15]

The focus of this thesis is short-term collaboration as defined above. The collaborative decision process in our case are meetings in consortiums of organizations and the aim of traceability, transparency and clarity is directly applied to the decision log of those meetings through distributed ledger technology.

1.3 Aim

This work aims to investigate the feasibility of a log of causally dependent events which is distributed both in terms of data storage and in terms of ordering and validating events, for the purpose of being used as a *secure*, *trustworthy* and *usable* log of decisions taken during large meetings within a consortium of organizations who do not necessarily trust each other. This is done by implementing and evaluating a prototype system based on a distributed ledger. The events that should be decided upon by the system and logged are decisions, votes and attendance. In this context:

- *Security* is provided by the log being immutable and tamper proof;
- *Trust* is provided by the distribution of the deployed software, the data and the modification of data among the consortium's organizations so that no central authority has to be trusted;
- *Usability* is defined in terms of response time and delay; users should experience a near real-time response to actions in the system and no significant delay should be added to the meeting as a whole.

Decision propositions should be presented to hypothetical or simulated clients for a certain amount of time before being logged along with attendance and votes. The votes should reflect the attendants that have reservations against the decision, and follow the model of *silent accept*; objections are explicit and acceptance is implicit.

Events relating to these decisions have to be agreed upon and ordered by the distributed system and stored in a way so that the information is immutable and shared between the organizations represented at the meeting. While providing these features, the delay added to the meeting as a whole should be as low as possible and response times for individual meeting attendants should be near real-time to ensure

usability.

The number of intended meeting attendants is related to the intended use of the system and a higher capacity of the system in terms of the number of attendants or organizations would make it applicable in more situations. The system has to support at least tens of meeting attendants to be usable at all, but the goal is for the system to be much more scalable than that, and support hundreds of attendants.

1.4 Delimitations

This work is limited to the decision log itself; how events are created, agreed upon and stored in a way that fulfills the requirements of trust and security. There are several factors that would be part of a system based on this work, but falls outside its scope:

- *Authentication.* There are many ways of handling client authentication in centralized or distributed ways. In this work, meeting participants will be assumed to be authenticated in some way, but the specifics are not relevant to the study.
- *User experience.* This study will talk about usability in terms of performance, not how participants would interface with the system.
- *Human communication.* When discussing remote meetings and meeting participants, the focus will be the capacity and usability of the decision log itself, not details regarding how the meetings themselves are conducted or scaled up.

1.5 Problem Specification

The evaluation and performance goals will have to do with the use case; memory or computational performance are not likely to be issues while delay in the system is critical, both for usability but also from a perspective of cost since meetings in industry often cost a lot of work hours and therefore money. Long delay times on transactions are also the typical drawback of distributed ledgers, and a high throughput of causally ordered events is difficult in any distributed system since the number of messages required in most sufficient consensus methods increases quickly when the number of nodes is scaled up.

The key question that this work will attempt to answer is: how large is the cost in time and usability for the proposed solution to the issues of trust and security, and at which point does that cost become too large? To answer this, the following metrics are defined:

- **Response time.**The response time describes the time between when a user presses the button to vote against a decision and when confirmation is given to the user that the vote is registered in the system. This metric tells us how much the consensus requirements affect usability from the point of view of the individual users. The users of the system would expect a near real-time response to their actions and even though joins, leaves and disconnects are causally unrelated, recording attendance for each decision implies that they

are causally related to, and must be ordered in relation to, other events of a decision.
- **Overhead**. The overhead describes the surplus time it takes for the system to go through the process of recording a decision during a meeting, in addition to the valuable time participants are given to cast their vote. This metric relates to the cost of the prototype system in terms of time added to a meeting.

1.6 Outline

Chapter 1 introduces the thesis. The issue and the proposed solution are presented along with relevant context. Finally, a concrete aim and the limitations of the thesis are given and a problem specification describes how the results are defined so that they relate to the aim.

Chapter 2 gives a background to the thesis by presenting theory relevant to the prototype design and implementation and to the experiment. This includes permissioned distributed ledgers in the context of the experiment and in particular the relevant theory behind the distributed ledger framework selected for the prototype system, and also theory regarding distributed consensus methods which gives context to the results.

In chapter 3, an abstract overview of the prototype system is presented where the different high-level components of the system are described and justified in relation to the experiment. Chapter 4 describes the implementation of the prototype system in enough detail for the experiment to be recreatable and for the results to be understood and related to specific implementation details.

Chapter 5 presents the experiment. Concrete definitions of the metrics, how they relate to the aim and how they are measured in the prototype are given and the results of the measurements are presented. Chapter 6 discusses the results in detail in relation to the work as a whole, and in particular to the prototype system and the expected results. Ethics and sustainability factors relating to the thesis are also discussed.

Chapter 7 presents related works and how they compare to this thesis in terms of aim, methodology and results. Finally, chapter 8 concludes the thesis and summarizes the aim, methodology, and findings.

2

Background and Choice of Framework

This work's application of distributed ledger technology to decision logs of meetings in the form of a prototype system builds on top of existing theory in a few key ways, all of which will be described below in this chapter.

First of all, a permissioned distributed ledger is chosen in favor of a permissionless system like Bitcoin or Ethereum that are mentioned in section 1.1. Specifically, *Hyperledger Fabric* is chosen. This choice aims to remedy the issue of the relatively impractical proof-of-work consensus methods commonly used in permissionless systems, but is also a natural choice in general when considering that the purpose of the decision log is to be shared among a consortium of organizations but not outside that select group of organizations.

Secondly, the consensus methods that are relevant for permissioned distributed ledgers in general and Hyperledger Fabric in particular are presented and discussed in terms of their pros and cons for a system such as ours. Specifically, they are compared in terms of guarantees and time complexity. Ideally, the consensus implementation provides Byzantine fault tolerance while still being efficient enough (defined in Section 1.5 and evaluated in Section 5.2) to work in our use case.

2.1 Permissioned Distributed Ledgers in General

A distributed ledger is a ledger, i.e., a database, that is replicated and distributed among members of a network. The replicas are synchronized by some form of consensus mechanism. The term is related to, but more general than, blockchain, which also defines that the data is grouped into blocks and cryptographically linked to each other to ensure immutability, as described in Section 1.1.

Blockchains and distributed ledgers are most often thought of as open networks with the proof-of-work consensus method [2] ensuring consensus among the peers while allowing anyone to participate, which is suitable for something like a currency. The method works well enough for financial transactions, but it is very slow and extremely computationally expensive compared to other methods of consensus. When all peers in the network are known and authenticated, we no longer have a need for some consensus method that allows public participation and more conventional consensus methods can be used, which would allow consensus to be reached much faster.

A permissioned distributed ledger is simply a distributed ledger where, rather

than allowing anyone to participate as in most cryptocurrency systems, only authenticated clients can access the ledger, which completely changes the suitable use cases. For this work, the ledger will contain information about closed meetings and we explicitly want the ledger to be shared and available within the consortium of organizations only, so a permissioned distributed ledger is suitable.

2.2 Hyperledger Fabric

Hyperledger is a project started by the Linux Foundation in 2016. As stated on the Hyperledger website:

> Hyperledger is an open source community focused on developing a suite of stable frameworks, tools and libraries for enterprise-grade blockchain deployments [14].

The largest codebase within Hyperledger is *Hyperledger Fabric*, or simply *Fabric*. Fabric was chosen for the implementation of the prototype system used in this work for a few different reasons, which will be presented in detail below. First, the modular nature of Fabric makes disregarding components that are irrelevant to this study, most importantly the specifics of the membership mechanism in the permissioned system, possible. Secondly, and most importantly, the *Execute, Order, Validate* paradigm of executing smart contract transactions used in Fabric allows the consensus mechanism to be implemented as a separate modular component. This greatly simplifies the process of comparing different methods of consensus, since no other component relies on the specifics of the method but rather just the fact that consensus eventually is reached.

2.2.1 Design Concepts and Motivations

The design choices of Fabric are in large motivated by drawbacks of existing commonly used permissioned distributed ledgers, in particular:

- Hard-coded consensus, limiting the possibility of using different consensus methods for different implementations with different needs and making other components of the system like transaction validation and access control directly dependent on the specific consensus method used.
- A domain-specific smart contract language is required due to the transaction model where transactions are agreed on first and executed later, which means that the transactions themselves have to be deterministic and always be executed sequentially by all peers [1].

To overcome these limitations, a few design choices are made which also makes Fabric very suitable for this work. The architecture is modular which means that different functions of the system are separated and interchangeable. In practice, Fabric is implemented as a set of Docker images performing different functions. This allows focusing on the components that are relevant for this thesis such as the smart contracts and the consensus implementation while easily disregarding what falls outside the scope, like membership service providers and certificate authorities.

Smart contracts can also be written in standard programming languages due to the unique way in which transactions are handled (see Section 2.2.2) which increases

performance by allowing parts of the execution to be run concurrently and makes the learning curve for development much less steep.

Below, some key concepts in the architecture of Fabric and how they relate to concepts in the use case of this thesis are presented.

2.2.1.1 Organizations, Consortiums and Channnels

The concepts of *organizations* and *consortiums* in Fabric are directly analogous to the concepts of the same names in this work.

An organization in Fabric is an entity in the network that through a membership service provider provides identities and access to the network for all other parts of the network. The network is made up of organizations and all modular components belong to an organization. This means that in our case, actual organizations can participate in the permissioned network on equal terms by deploying their own components, including their own membership service provider for managing access.

A consortium in Fabric is simply a group of organizations that can communicate with each other and share one or more ledgers. In our case, they will share ledgers used for meeting decision logs. For each consortium, a *channel* is defined which, as the name states, is the channel of communication for the consortium. Since there is a one-to-one relationship between channels and ledgers, a channel will define a meeting in our case. For each meeting, the consortium of organizations that will participate creates a new channel and the channel's ledger defines the decision log.

2.2.1.2 Smart Contracts and Chaincode

Smart contracts are implemented in Fabric as *chaincode*. The terms are used interchangeably even though smart contracts formally defines the transactions and chaincodes are specific Fabric implementations packaging smart contracts [30], but for the purpose of this work, we will use the more general term *smart contract*.

First of all, *assets* are defined as the stateful targets of transactions in the ledger. The assets can be implemented as any objects in the supported languages and are saved in the ledger as key-value pairs. The ledger itself is a blockchain holding the record of all state changes on assets. Secondly, *transactions* define how the state of assets can change and are implemented as methods in the smart contract that can be invoked by clients. Finally, each smart contract has a defined *endorsement policy* that simply specifies the set of *endorsing peers* for the contract; the peers that have to *endorse* a certain transaction for it to be considered valid by the network. Why this is necessary and the process of how this is done is described in the following section.

2.2.2 The *Execute-Order-Validate* Paradigm

When using the model of first ordering transactions and then executing them sequentially on all replicas of the distributed ledger, consensus about the order of state-changing transactions is not sufficient to ensure that all replicas of a ledger maintain the same state; if the transactions themselves are non-deterministic, the replicas might still diverge. To remedy this, other distributed ledger systems that

rely on the paradigm of ordering transactions first and and then executing them sequentially uses a deterministic domain-specific language for smart contracts.

To allow the use of general-purpose languages in smart contracts, Fabric uses another transaction model called *execute-order-validate* [1]. First of all, the transaction logic and the consensus mechanism that would otherwise both be part of the ledger replicas to ensure that all replicas are in agreement about the ordering of transactions are split into two different components; *peers* and *orderers*.

The peers implement most of the functionality of the system and are the endpoints of clients interacting with the ledger. They hold the actual ledger itself, a state database keeping track of the current state of all assets, and the smart contracts that define the assets and transactions. The orderers have the sole purpose of reaching consensus about the order of transactions. They are completely decoupled from the specific ledger and contract implementations and simply use some distributed consensus algorithms among themselves to decide and report back the order of arbitrary transactions sent their way.

2.2.2.1 Execute

The first phase of the transaction paradigm, *execute*, is done by the peers. Recall that the ledger data is in the format of key-value pairs. After some client invokes a transaction of a smart contract kept by a peer, all peers defined as endorsing peers in the endorsement policy simulate the transaction and creates two sets of key-value pairs:

- The *write-set* contains all keys that are modified if the transaction is eventually validated and added to the blockchain along with their new values.
- The *read-set* contains all keys that were read during the transaction simulation when producing the write-set, along with their version number in the blockchain so that other peers can look up their values.

The sets of key-value pairs produced are then signed, or *endorsed*, by the peer and sent back to the client that initiated the transaction. The peers defined by the endorsement policy must not only answer with a signed response, the response must also be the same for all peers for the transaction to proceed to the next phase. Requiring all peers defined in the endorsement policy to simulate the executing of the contract and come up with the same hypothetical state changes and dependencies (i.e., the write- and read-set) before ordering is the core of how non-deterministic smart contracts are tolerated.

2.2.2.2 Order

Order, the second phase of the transaction paradigm, is done by the ordering service consisting of several separate orderers and invoked by the client that initiated the transaction after the transaction has been endorsed by all peers specified in the endorsement policy. The ordering service consists of a set of nodes owned by separate organizations that run some distributed consensus algorithm to totally order all transactions. The ordering service also groups transactions into blocks and cryptographically chains those blocks to form the actual blockchain. When consensus is

reached between the ordering nodes about the order of submitted transactions, the finished block is sent to all peers in the channel for validation.

There is no dependence between the method used to order the transactions and the endorsement or following validation of the transactions so any distributed consensus algorithm that totally orders events could be implemented. This is the part of Fabric that is called *pluggable consensus* [1]. In practice, officialy implemented alternatives are still only crash fault tolerant at best [18]. A proof-of-concept wrapper around the Byzantine fault tolerant library BFT-SMaRt that suffices in demonstrating the added time complexity of handling Byzantine faults also exists [3]. Further background regarding distributed consensus is given in section 2.3, and regarding Byzantine fault tolerance in particular in section 2.3.2.

2.2.2.3 Validate

Validate is the third and final step of the transactions paradigm and is done by each peer after a block of executed and ordered transactions is received from the ordering service. Since the ordering service cryptographically chains the blocks that make up the blockchain stored by the peers, all transactions will be added to the blockchain regardless if they are deemed valid or not, but the current state will only reflect valid states. Each peer checks for validity in two ways and after that a bitmask reflecting the result of these checks marks the transactions as valid or invalid in the blockchain.

The first check is done in parallel for all transactions in the block and simply checks for signatures from all endorsing peers in the same way the initiating client checked for signatures in the *execute* phase.

The second check detects read-write conflicts and is therefore done sequentially for all transactions in the block. Recall that each transaction is already simulated by each peer in the *execute* phase, which produced a read-set and a write-set. Now, for each transaction, its read-set is compared to the current ledger state and the write-set is written if they match. If the read-set does not match the current state of the ledger, it is marked as invalid and added to the log of transactions without its write-set affecting the current state of the ledger so that race conditions due to concurrency are prevented.

2.3 Distributed Consensus

There are many options to choose from when it comes to consensus methods without the restrictions introduced by a permissionless system, and each has their advantages and drawbacks. Simpler algorithms might be simpler to implement and more efficient in terms of the number of messages needed to reach consensus, but might not provide the same guarantees as more complex algorithms.

Consensus methods with a certain level of fault tolerance are usually similar in theory and complexity but can vary significantly in implementation.

In this section, background relevant to consensus methods commonly used in permissioned distributed ledgers in general and the methods used in this study in

particular is given, with a focus on the differences between a crash fault tolerant (CFT) and a Byzantine fault tolerant (BFT) method.

2.3.1 Crash Fault Tolerance

Crash fault tolerance, in terms of distributed consensus, is ensured by a protocol where consensus is eventually reached even in the presence of faulty processes. Some important assumptions made about the system for a CFT protocol to be applicable and suitable are:

- All correct processes answer within some time frame or are timed out. In an asynchronous system, consensus is proven to be impossible to solve even in the presence of one faulty process [11]. When waiting for a process, it is impossible to know if it has crashed or if the response is simply delayed, so the system has to be made semi-synchronous in some way. A common way of handling this is to define a process that takes too long to respond as faulty and disregard it, even if it might answer eventually.
- All working processes follow the same protocol. To handle potentially malicious processes, not just working or crashed ones, completely arbitrary behavior must be accounted for and the consensus mechanism becomes more complex.

2.3.1.1 Paxos

One common way of solving consensus is algorithms based on Leslie Lamport's Paxos algorithm [16]. The Paxos algorithm solves fault tolerant asynchronous consensus, so messages might be lost or arbitrarily delayed, but it is assumed that all agents actually follow the protocol and that no messages are corrupted. Google uses a Paxos-like algorithm in their distributed lock service *Chubby* and notes that "all working protocols for asynchronous consensus we have so far encountered have Paxos at their core" [5].

As Lamport notes himself in his 2001 paper, *Paxos made Simple* [17] (Paxos is the name of a Greek island):

> The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers. In fact, it is among the simplest and most obvious of distributed algorithms. [17].

Several attempts to explain the algorithms more clearly than in the original paper exists, including Lamport's own attempt. In short, the algorithms uses two rounds of message passing for each value that should be agreed on, *prepare* and *accept*. Processes acts as *proposers, acceptors,* and *learners*. In the first round, proposers propose numbers and seek accepts from a majority of acceptors, which only accepts numbers higher than any previous number. In the second round, the actual value is decided by a quorum of acceptors and sent to learners. Leadership election is used to select a distinguished proposer as the only one issuing proposals to ensure progress [17].

The Paxos algorithm is often implemented as *multi-Paxos*, which works as Lamport's original description of Paxos but with one important optimization. If a suc-

cessful leader election has occurred and a distinguished proposer exists, it will always propose increasing numbers and the first round becomes unnecessary as long as that leader is still working [33].

2.3.1.2 Raft

Raft is a consensus algorithm that aims to offer the same guarantees and similar performance of Paxos while being easier to understand and implement. The default ordering service of Hyperledger Fabric since version 1.4.1 is based on Raft [18].

Ongaro et al. [23] highlights the perceived flaws with Paxos and how Raft attempts to remedy them:

- Paxos was designed with was is called *single-decree* as its foundation, as opposed to multi-Paxos (described above), even though multi-Paxos is more practical in most cases. Indeed, reaching consensus on a single decision is the point of view of Lamport's description in *Paxos Made Simple* [17]. This is pointed out as the reason for the lack of actual open-source implementations strictly based on Lamport's works and for the perceived difficulty to understand and describe the algorithm. On the other hand, Raft was designed as a log replication system from the start, rather than a system where a set of entries are agreed on that then must be melded into a log. Several full-specification open-source implementations of raft exist [27] and a user study was conducted which indicates that Raft is easier both to understand and implement than Paxos.
- Raft differentiates itself from Paxos by the use of a stronger leader node. It is argued that in a real-world scenario where multiple decisions have to be made (in contrast to the reasoning behind single-decree Paxos), and since a leader election still has to be performed, it is more practical and efficient to first elect a leader and then let that leader coordinate following decisions.

2.3.2 Byzantine Fault Tolerance

Another family of algorithms for solving consensus in distributed systems are ones that are Byzantine fault tolerant (BFT). Similar assumptions about the system are made as in the CFT case, but with one major difference; processes are not assumed to be following the protocol correctly. Instead, BFT algorithms aim to continue working correctly even when a subset of processes act completely arbitrarily. In the 1980 paper *Reaching Agreement in the Presence of Faults* by Lamport et al. [25], it was shown that the problem stated in the title can be solved:

> "for, and only for, $n \geq 3m+1$, where m is the number of faulty processors and n is the total number". [25]

2.3.2.1 Advantages and Disadvantages

Byzantine fault tolerance is especially relevant in the case of permissioned distributed ledgers since different nodes in the system are controlled by different organizations. In a distributed system where one party owns all nodes, the code running on each node is controlled and the essential part of fault tolerance is to handle the unknown factors of the system; network faults and crashes. In the case

of distributed ledgers though, different parties control the code running on different nodes and making sure that some nodes cannot maliciously control the outcome of transactions or the state of the system is a much more relevant concern.

The downside of Byzantine fault tolerant algorithms is an increase in the number of messages and message passing rounds required to reach consensus. While a CFT algorithm assumes that all nodes actually try to reach consensus, a BFT algorithm has to take into account nodes that actively try to prevent consensus from being achieved. For example, they might try to manipulate the system by lying about which messages they have received or by sending different responses to different nodes. To be able to mask this broader range of faults, a more complex algorithm is needed.

2.3.2.2 Practical Byzantine Fault Tolerance

A practical BFT algorithm was first proposed by Miguel Castro and Barbara Liskov in 1999 [7]. The algorithm defines a *view* as the set of replicas in the system and assigns a single *primary* replica based on the replica's ids. Messages in the consensus process between replicas are *accepted* based on a digest and sequence number of a message and the current view visible to the replica. There is a protocol in place to handle view changes that will not be discussed in detail here.

In short, during normal-case operation, the algorithm uses three rounds of message passing between a request to the primary and a reply; *pre-prepare, prepare,* and *commit*. When the primary receives a request, it sends *pre-prepare* messages with a certain sequence number to all replicas. If the message is accepted by a replica, it sends *prepare messages* to all other replicas, including the primary. When a replica has received $n \geq 2m$n valid *prepare messages* including its own, where m is the maximum number of faulty nodes (see Section 2.3.2), it sends a *commit message* to all other replicas. If a replica has received $n \geq 2m + 1$ valid *commit messages* including its own, it sends a reply to the client [7].

Essential to this work is that, as can be seen in Figure 2.1, messages are multicast by all replicas except the primary in the prepare phase and all replicas in the commit phase. This means that when increasing the number of nodes, the number of exchanged messages will increase much faster for the pBFT algorithm than for Paxos.

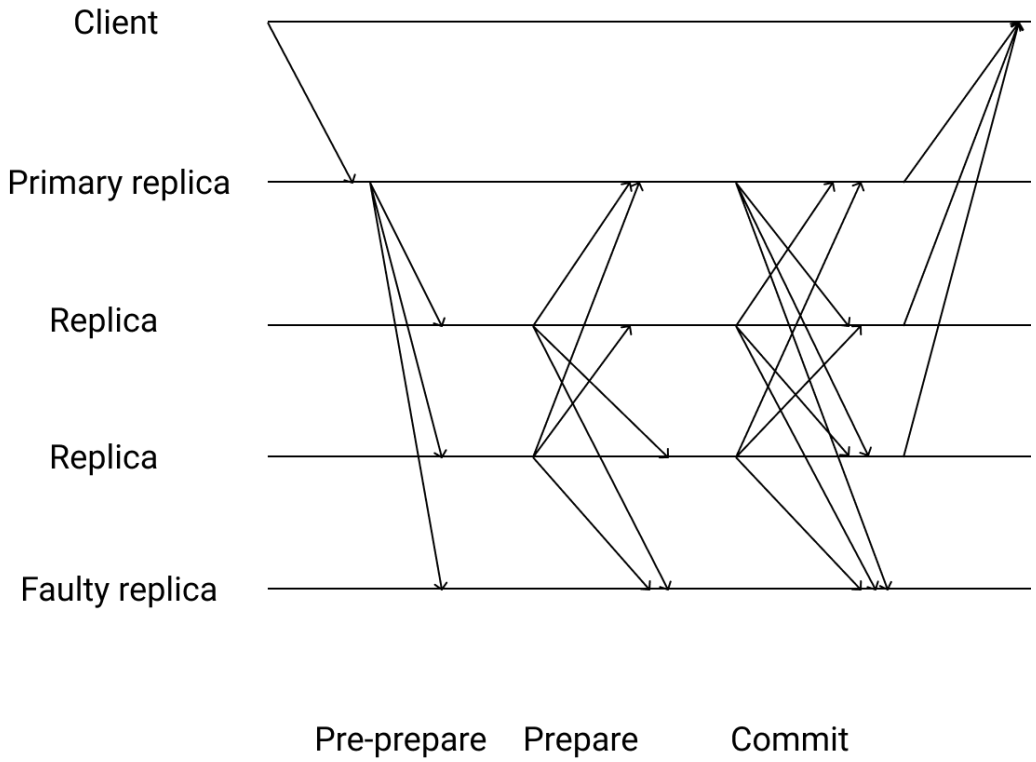


Figure 2.1: The messages passed between replicas in normal-case operation with four total replicas, capable of handling a single faulty replica.

2.3.2.3 Byzantine Fault Tolerance in Permissioned Distributed Ledgers

There are many algorithms that are based on pBFT, some of which are used in permissioned distributed ledgers. One example is the Istanbul BFT consensus algorithm [20] which is used in the permissioned blockchain platform *Quorum*. Another example is an ordering service for Fabric implemented as a wrapper around BFT-SMaRt [3], which is a Byzantine fault tolerant state machine replication protocol implemented as a Java library.

Using a BFT algorithm in the prototype system of this thesis further increases the guarantees that are provided by the system to include arbitrary or malicious activity from some organizations in the consortium. However, it comes at the cost of more message passing rounds needed to reach consensus and a higher message complexity in terms of the number of nodes, which might present an obstacle when scaling up the number of organizations and participants at the meetings.

3
Prototype Design and Overview

This chapter describes the design choices made for the prototype system and gives a high-level overview of its functionality and components.

3.1 Design Motivations

The design of the prototype system was decided on based on the aim of the system as secure and trustworthy by implementation and with a quantifiable usability, along with the context given by Yolean providing domain-specific information about meetings in consortiums in terms of functionality and scale. The following four aspects guided the prototype system design:

- *Similarity to a production system*
- *Distribution*
- *Lightweight user client*
- *Scalability*

The prototype system is intended to act as a proof of concept for actual production systems, so it should be *similar to a production system* so that both performance metrics and the system as a whole can be evaluated in the context of such a system. There already exist several performance evaluations of permissioned distributed ledgers from a generic point of view [13, 32, 22]. In contrast, the performance evaluation of our system is done in the context of a specific use and aim; a distributed decision log and its usability.

To provide trust as security as defined in this work (Section 1.3), the system should be completely *distributed* among the consortium's member organizations. Each organization should run a replica of the distributed ledger to have a stake in the storage of and transactions on the data, and all other parts of the system should be deployed separately for each organization and only get and modify the meeting state through the ledger, so that no organization has to rely on software deployed by another organization.

Each participating organization will run a replica of the system as described above, but for the system to be usable and practical in actual meetings of different scales, the *user client should be lightweight* and simple to use and deploy. The individual users of the system during a meeting should be able to participate using something as simple as a web interface or a smartphone application, and all business logic should be handled by the distributed system and not rely on the participating clients.

For the sake of functionality and ease of deployment and use in different con-

sortiums of different sizes, the prototype system must be easily *scalable* in terms of participating organizations. For the experiments and evaluation of the system, the system also has to be easily deployed in a repeatable way at different scales in terms of the number of ledger replicas, the network delay between the components and the number of participating clients.

3.2 Decision Process Model

The prototype system represents a meeting as a log of decisions which are presented to users sequentially according to a meeting protocol or similar. Following this, decisions can be in three distinct phases; *before, during* or *after* they are handled by the participants of the meeting. When a decision is voted on, the *silent accept* model is followed, so attending users either implicitly accept a decision or explicitly record their reservation.

To be able to keep a record of who took part of information during a meeting, attendance must be asserted by the system every time new information about the decision arrives, which is every time the decision enters a new phase. A user can be said to have accepted a decision only if he was in attendance at the start and end of the voting phase without explicitly objecting.

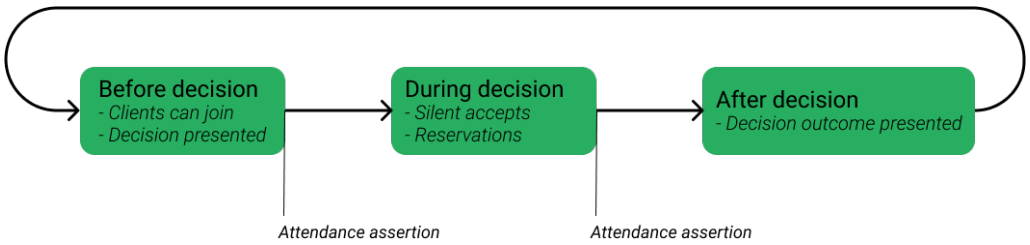


Figure 3.1: The decision process

The Smart Contract: A smart contract defines the stateful information stored in the distributed ledger on which the system is based in terms of *assets* and *transactions*. In our case, a ledger belonging to a consortium represents a meeting and the assets logged in the ledger are decisions. Following the decision process model, the state of these assets is defined by:

- the current *phase* of the decision
- a list of the current *attendants*
- their attendance and,
- their potential *reservation*

The transactions define the following state changes of assets:

- changing the phase of the decision
- adding and removing clients from the attendance list
- setting the attendance of clients
- setting the reservation of clients

These transactions also define under which conditions the state of an asset can change in a certain way. The phase can only change after attendance is asserted and

in the order outlined in Figure 3.1, clients can only be added to the attendance list before the decision and clients can only set their reservation during the decision.

3.3 Adversary Model

To fulfill the trust requirement of the system and ensure that the backends can run in parallel without giving any organization the possibility of changing the business logic or lie about its clients in any meaningful way, it is important that all events related to the meeting state result in transactions that are agreed on by the network of ledger replicas. In particular, the backend of a malicious organization might lie about:

- The amount of information received before voting
- The amount of time passed during the voting phase

Let us look at how to ensure the properties above. An organization might want to claim that it received less information than it actually did. Logging the attendance at the start of and end of the decision process prevents this. To participate in a decision, a backend must assert the client's attendance. If it acts unresponsive later in the decision process, the initial attendance has still been immutably recorded and it can be shown that the client stopped responding after receiving the information.

Asserting the time passed is a bigger issue due to the asynchronous nature of networks. Each time one organization tries to end the voting phase of a decision, another might claim the transaction to be invalid since not enough time has passed, which might be true due to network delays or other factors. Malicious intent might be assumed if these specific transactions are consistently delayed while others are not, and if all transactions are delayed enough to give attendants significantly more time to vote (in the order of seconds), that organization might be excluded due to the high response time and cost added to the meeting. However, no ways of handling these kinds of attacks are currently implemented in the prototype system.

3.4 Components

The prototype system consists of three main components; the distributed ledger, a backend, and the clients. Each participating organization runs a system consisting of two components; a backend and a ledger replica, and individuals participating in a meeting connects a client to the backend component of their organization.

In this way, each organization can run the complete system in their own cluster (or however else they wish) and their user clients only connect to their own backends, so that they do not have to depend on other organization's deployed software. All events that change the state of a decision in any way result in a transaction to the ledger, so all cross-organization communication is confined to the ledger replicas reaching consensus.

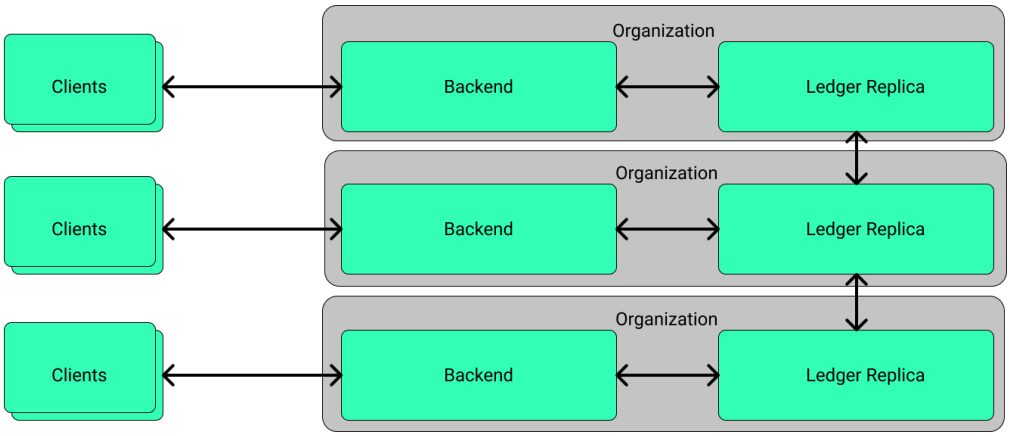


Figure 3.2: An overview of the system components

3.4.1 The Distributed Ledger

The distributed ledger component is made up of the ledger replicas of all participating organizations, and each organization's ledger replica provides access to the ledger to the organization's backend component. The intended use of the system is for each organization to run their own ledger replica so that both the validation and ordering of events and the data storage itself are distributed across all parties.

All ledger replicas run a smart contract defining the assets and transactions. In our case, the assets represent decisions during a meeting and the transactions define all possible state changes that a decision can undergo. The transactions are invoked by backends of the participating organizations and the distributed ledger decides the validity of the proposed transactions and reaches consensus about their order as described in Section 2.2.2.

3.4.2 The Backend

The backend is the endpoint for user clients to act on the system, so it interfaces with clients on one end and with the distributed ledger on the other. The logic between the clients and the ledger decides the decision process and abstracts the data stored in the ledger and the possible state changes, defined by the smart contracts as started above, into client actions.

Some actions, like clients joining a meeting or reserving their vote on a decision are initiated explicitly by the clients and others, like a client timing out or silently accepting a decision, must be inferred by the backend based on the state of the ledger and the connected clients.

Many instances of the backend can work concurrently since the distributed consensus regarding order is solved by the ledger, so ideally each organization would run their own instance of a backend to remove any central point in the system and provide the trust as specified in Section 1.2.

3.4.3 The Client

The clients interface with a backend to provide the explicit actions and implicit states mentioned above. For the purpose of this work, the client component could hypothetically be omitted in favor of the backend simulating events directly, but for the sake of separation of concern and also for the sake of making the prototype system as similar to a hypothetical production system as possible, clients are implemented as a separate component.

3.5 Kubernetes for Deployment and Scaling

Kubernetes [4] is an open-source container-orchestration platform developed by Google. Since the Hyperledger Fabric is implemented as a set of Docker images, a suitable way of implementing and deploying the rest of the system was to containerize all parts and deploying them in a Kubernetes cluster. It is not strictly necessary to run the prototype system in a Kubernetes cluster, the same results could be achieved on a single host using Docker Compose, but Kubernetes has a few advantages:

- *Ease of deployment and scaling.* There are many different ways of building containers and deploying them to Kubernetes. Which ones that are used for different components in the prototype system is discussed in detail later in this chapter. When deploying the entire system to a Kubernetes cluster, scaling components and parameters up and down is often as easy as changing a few lines of markup, where more specific changes for individual containers would be needed using Docker Compose.
- *Jobs.* The concept of jobs in Kubernetes makes it easy to create consistent deployments and configurations. Setting up a complete Hyperledger Fabric system requires a considerable amount of configuration and is usually done using a number of bash scripts, as can be seen in Fabric's sample Github repository [10]. This would not be such a major issue in production systems where, for example, a set of peers for an organization are set up once and then used during the lifetime of the system. However, in our case, we want to reproduce configurations with minor changes very frequently to run test cases and collect metrics, which makes the initial setup much more tedious. The setup process can be simplified immensely by using jobs and related open-source tools.
- *Realism and re-usability.* In an actual production system, some container-orchestration system would probably be much more likely than specific deployments for individual hosts. In fact, Yolean uses Kubernetes and many of the tools and procedures used in this work for their own software deployments. For future works or implementations based on this work, using Kubernetes makes it much more likely that code will be reused.

The nodes in a Kubernetes cluster are called *pods*, and they encapsulate one or more containers into a single deployable unit. Pods deployed in a Kubernetes cluster are logically separated from each other and assigned unique IP-addresses while the containers running in a pod share some resources and communicate with each other

over `localhost`. In our case, the pods each represent a distinct component in the system. The orderer and backend pods run one container each while the peer functionality is made up of four different containers. Each organization runs their peers, orderers and backend in a Kubernetes cluster (Figure 3.3).

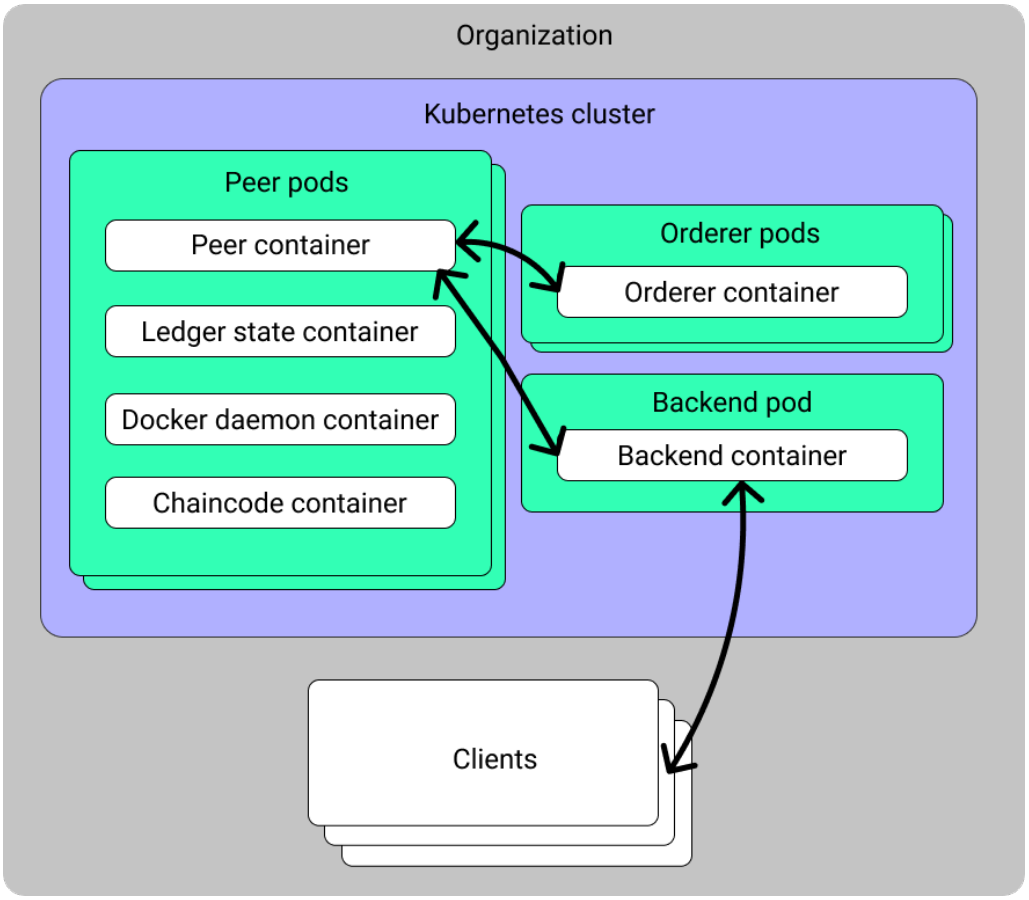


Figure 3.3: The components of the prototype system with the default (Raft) ordering service.

Helm Charts: Helm charts provided by the open-source project *PIVT* [26] are used for deploying all components of Fabric. Helm is a tool for managing applications in Kubernetes; a *chart* defines the components of the application based on *templates*. Each component template loops over a set of component definitions passed to it in a markup file at deployment and substitutes specified values in the configuration based on those definitions.

Official Hyperledger Fabric sample repositories rely heavily on complex shell scripts and individual configuration files for each component, which makes reconfiguration complex, especially due to the separation that is introduced by running the system in a Kubernetes cluster. Helm allows us to easily recreate similar components with minor differences in configuration like peers and orderers and to easily scale up the number of said components by using template files.

The Helm charts provided by PIVT deploys peers and orderers as pods in the Kubernetes cluster. Argo Workflows, a workflow engine used to set up jobs in Kubernetes, is then used to create jobs for setting up channels on consortiums and

installing smart contracts on those channels according to a configuration file. The charts also sets up membership service providers and certificate authorities for all organizations and joins all components to the appropriate organizations. This is very helpful when scaling up the number of peers and orderers but is not discussed in detail as it falls outside the scope of this work.

The orderer pods simply run one container each; the orderer itself. The peer pods run:

- A peer container which keeps the blockchain itself in local storage and handles the logic of the peer as described in Section 2.2;
- A ledger state container running a database that, generated from the blockchain kept by the peer, provides the current world state and (for performance as each transaction validation looks up previous states) an indexed view of all transactions;
- A chaincode container running the installed smart contracts;
- A Docker daemon container, running Docker-in-Docker. The peer container needs an endpoint to a Docker daemon to set up the chaincode container on the pod. Current versions of Kubernetes use *containerd* as the container runtime rather than Docker, so a separate Docker daemon is needed.

4
Implementation

This chapter describes specific choices made regarding the implementation of the prototype system, and the implementation in detail in terms of components and functionality. Since the essential part of the evaluation of the prototype system is the distributed ledger component and its performance, the system has been implemented with the intention of being easy to test and scale in ways relevant for the experiments.

4.1 The Ledger Component

The ledger component is implemented in Hyperledger Fabric as a number of organizations. An organization in Fabric is defined by a membership service provider, and the membership of an organization in a consortium along with its permission to create channels is controlled by the consortium's orderers. Since the orderers are also members of organizations, and for the sake of separating the membership service providers for these two purposes, each actual organization will in our case run two organizations in Fabric; one for the orderers and another for all other components.

It is assumed that all participating organizations want to run at least one peer and one orderer so that they can participate in both the validation and ordering of transactions, but in theory some organizations could run peers but not orderers or vice versa. An organization might want to run more peers for redundancy or, more importantly for this work, more orderers to tolerate a higher number of faulty or malicious orderers in other organizations.

It is also assumed that all peers are both committing and endorsing peers, each executing the chaincode for all transactions and holding a replica of the ledger. It is important that at least one peer from each organization endorses and validates each transaction proposal, and since each organization is assumed to have at least one peer, it is suitable for testing purposes to cover the worst-case scenario performance wise and require all peers to validate all transactions. Following this, the organization membership for peers is not essential when scaling up the number of peers for testing. In a production deployment, the delay would most likely be longer between the clusters of different organizations, but delay is already something we can add between peers, and can just as well be added between peers within an organization.

4.1.1 The Smart Contract

Smart contracts in Hyperledger Fabric are defined by assets stored in the ledger and transactions on those assets. The ledger has two components; the *world state,* which

contains information about the current state of the assets, and the *blockchain* which contains an immutable log of all transactions on said assets [30].

4.1.1.1 The Asset

As described in Section 2.2, smart contracts in Fabric are written in conventional non-deterministic programming languages. Our smart contract is written in JavaScript and defines an asset in JSON as follows:

```
decision_id: {
  state: 'pending' | 'ongoing' | 'ended',
  attendance: [
    {
      client_id: uid,
      hereAtStart: boolean,
      hereAtEnd: boolean,
      reservation: boolean
    },
    ...
  ]
}
```

The `decision_id` key is the key to the asset (decision) in Fabric's state database and the following object is the value.

The `state` describes the status of the decision. A decision is initiated as `'pending'`, set to `'ongoing'` during voting, and finally set as `'ended'` when the decision process is over.

The value of the `attendance` key is a list of all clients participating in the decision. Each client is identified by a `client_id` and has three additional boolean properties: `hereAtStart` indicating if the client acknowledged its attendance after the decision process began (decision state set to `'ongoing'`), `hereAtEnd` indicating if the client acknowledged its attendance after the decision was ended (decision state set to `'ended'`), and `reservation` indicating if the user voted against the decision.

4.1.1.2 The Transactions

The transactions of the contract are defined as follows:

```
startDecision(client_id): decision_id
join(client_id, decision_id)
here(client_id, decision_id)
reserve(client_id, decision_id)
setOngoing(decision_id): 'success' | 'error'
setEnded(decision_id): 'success' | 'error'
```

The `startDecision` transaction adds a new decision asset to the ledger state with a unique id and returns that id to the caller.

The `join` and `here` transactions are invoked by a client with a unique `client_id` to join a decision or announce its attendance. A decision can only be joined

if its state is `'pending'` and the attendance announcement sets `hereAtStart` or `hereAtEnd` for the client with matching `client_id` to `true` depending on if the state of the decision is `'ongoing'` or `'ended'`. The `reserve` transaction sets the `reservation` boolean for the client with matching `client_id` to true.

The `setOngoing` and `setEnded` transactions changes the state of a decision and returns the outcome of the transaction. Since the attendance announcements of clients depend on the decision state, a confirmation of the transaction is returned after which the clients can be asked to announce their attendance for the given state.

4.1.2 A Byzantine Fault Tolerant Ordering Service

The BFT ordering service implemented in this work, and currently the only available BFT ordering service for Fabric, is a wrapper around BFT-SMaRt (Byzantine Fault Tolerant State Machine Replication) by João Sousa et al. [31]. In the 2014 paper *State Machine Replication for the Masses with BFT-SMaRt* by João Sousa et al. [3], it is argued that even though a lot of academic work has been done about Byzantine fault tolerance, practical use in real deployments is lacking due to the lack of robust implementations. BFT-SMaRt aims to provide this by implementing an understandable and simple yet high performance library. The method of reaching consensus is roughly similar to practical Byzantine fault tolerance, as can be seen in Figure 4.1. The three phases (*propose, write, accept*) correspond to the phases as described by Castro and Liskov in 1999 (*pre-prepare, prepare, commit*) [7].

In the BFT-SMaRt Fabric ordering service, the architecture is similar to the Kafka-based implementation that was the default ordering service of Fabric before version 1.4.1 rather than the current Raft implementation. Rather than the service being completely made up of the ordering service nodes spread out over different organizations (Figure 3.3), organizations run *frontends* (Figure 4.2) that represent the ordering service nodes, while the BFT consensus is solved by a separate cluster of $3f+1$ ordering nodes where f is the number of tolerated faulty or malicious nodes (as described in section 2.3.2).

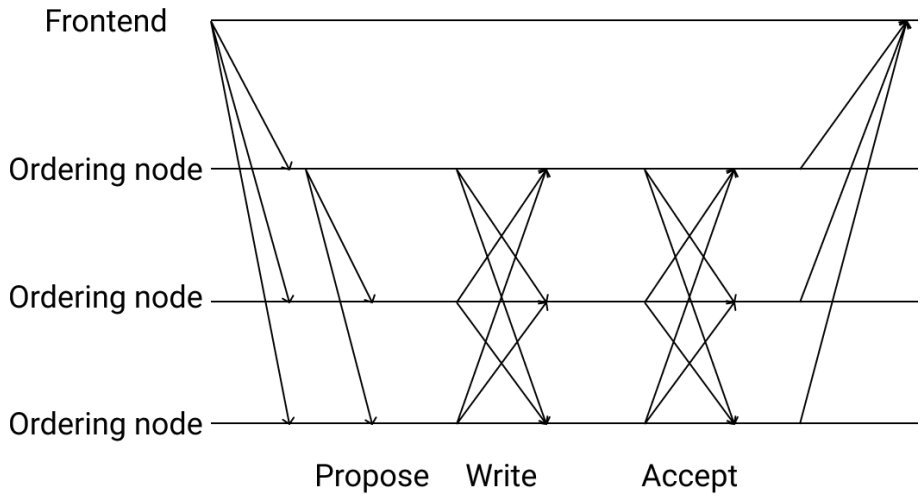


Figure 4.1: The message pattern of BFT-SMaRt [31] in normal-case operation.

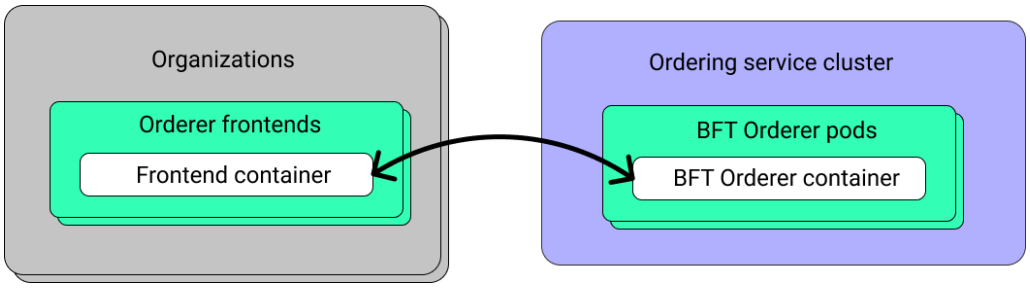


Figure 4.2: The components of the BFT-SMaRt [31] ordering service

This way of organizing the ordering service presents an issue in our case when it comes to decentralization, since we would like each organization to have an equal stake in the ordering without having to trust a separate network of ordering nodes (further discussed in the article *Demystifying Hyperledger Fabric ordering and decentralization* by Arnaud Le Hors [18]), and an official BFT ordering service is planned that will be more similar to the current Raft implementation to address the issues of decentralization (as stated in the Fabric documentation [24] and in this very informative mail exchange including developers of both Fabric and BFT-SMaRt [28]). However, the current BFT-SMaRt wrapper should be sufficient in highlighting the added performance cost of solving BFT consensus.

4.2 The Backend Component

The backend component has three distinct responsibilities:

- *Business logic.* As described in Section 4.2.1 the smart contract defines an asset representing a meeting and exposes several transactions that can be invoked on assets. However, the application itself is exposed to clients as a system of meetings and decision proposals. The business logic of our application is more complex than can be fulfilled by simply having clients invoke transactions, and this additional complexity is handled by the backend. This includes:
  - The business logic of the decision itself. After a decision is initiated, each client receives a certain time window from the backend during which it can choose to act. No further actions can be accepted from a client after its time window has passed and when the time window has passed for all clients, the decision should be ended.
  - Controlling client attendance, and most importantly deciding when clients are timed out. This is important since each decision depends on all participating clients, so an unresponsive client will add a cost to the meeting as a whole. It is also important to note that the order of events during decisions are causally related to the participation; no client leave should be logged after an event that the client did not see. Therefore, the attendance of all clients must be ensured at each related event.
- *Fabric Gateway.* The backend implements Fabric's Gateway SDK [12], acting as a gateway between the Fabric network and the business logic implemented between the backend and the clients.
- *Metrics.* The backend collects the metrics for the experiment (Chapter 5).

4.2.1 Implementation Details

The backend is developed in Node.js and deployed as a separate pod running a single container in the same Kubernetes cluster as the Fabric pods (Figure 3.3). Along with the backend pod, a Kubernetes *Service* resource is created, simply mapping the internal cluster address of the backend to an externally accessible address for communication with clients.

Using the official Fabric SDK for Node.js to communicate with the Fabric network, a transaction of the smart contract is invoked in the following way:

1. A given *connection profile* containing information about the Fabric network is read from the file system. The profile specifies the location of at least one peer to be used as the endpoint and one orderer (the organization's orderer frontend in the case of BFT, see Section 4.2.3) for accessing the ordering service. It also contains the location of and credentials for accessing the certificate authority of the organization.
2. If not already done:
   - Using the given credentials and certificate authority location, the backend enrolls a user for itself to the certificate authority and is given a X.509 certificate used to authenticate itself within the organization.
   - The X.509 certificate is saved in the file system to be reused for each successive transaction.
3. A *gateway* to the Fabric network is opened using the connection profile and X.509 certificate.
4. The channel and smart contract is specified and the transaction is invoked.
5. When a response to the transaction is received, the gateway is closed.

4.2.2 Client API

The following API is provided to the client:

- `/here`, announces its attendance.
- `/startDecision`, tells the backend to initiate the decision process.
- `/reserve`, votes against a given decision, rather than following silent accept.

The `startDecision` and `reserve` endpoints are called explicitly by a user while the `here` endpoint is automatically called by the client at startup to announce its existence to the backend. The backend keeps track of existing clients by keeping their addresses in a list along with a unique client id. When a new client announces its existence to the backend, it is added to the list and when it is unresponsive, it is removed from the list.

4.2.3 The Decision Process

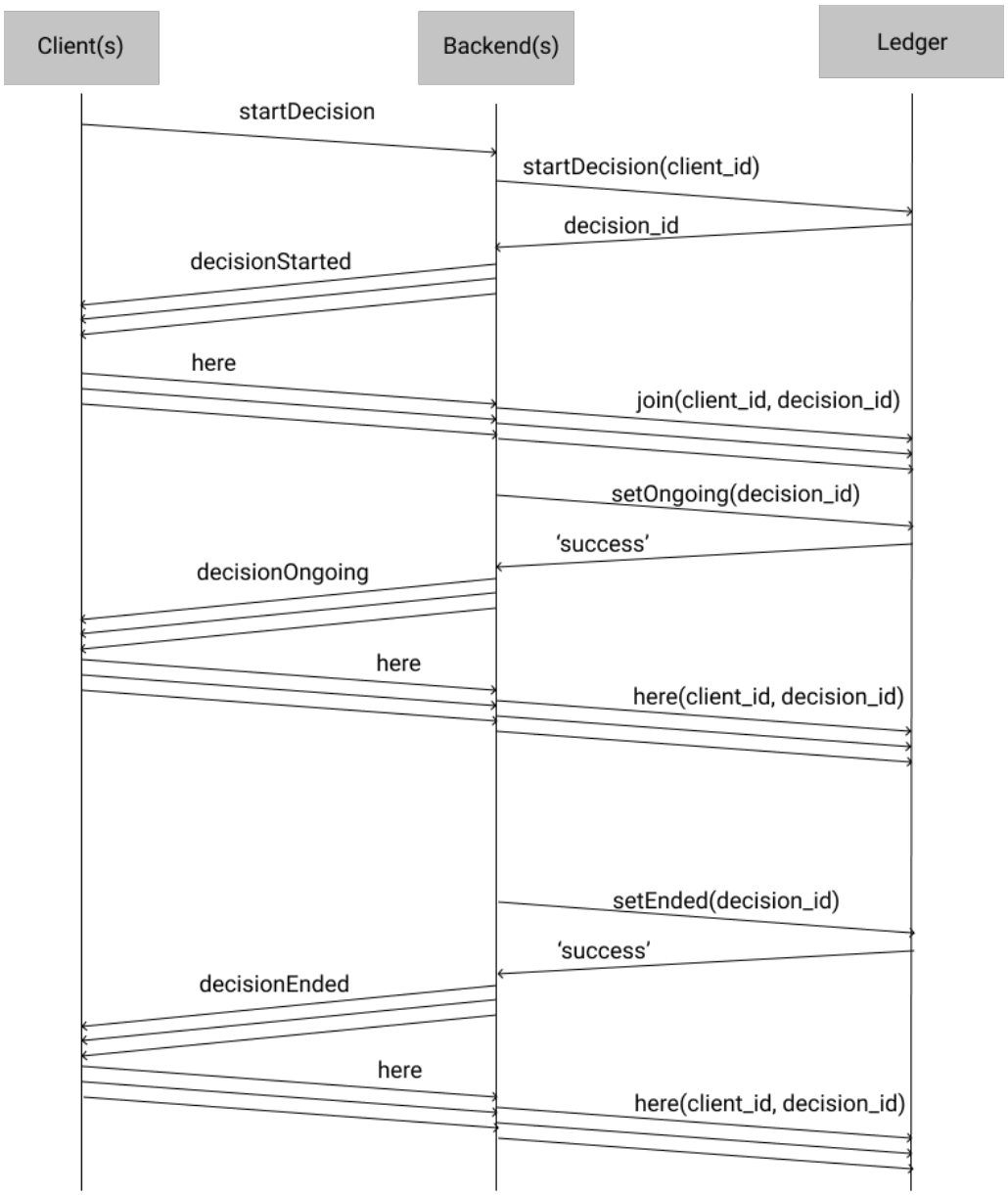


Figure 4.3: The communication between the components during the decision process.

The decision process (Figure 4.3) can be divided into three phases according to the three states of a meeting represented in the ledger:

- *Pending*: This phase is initiated when some client calls the `startDecision` endpoint on the backend, which:
  - creates a new pending decision and adds it to the ledger state. The smart contract responds with a unique id to identify the decision. The clients are notified and respond to join the decision, or are timed out.
  - invokes the `join(client_id, decision_id)` transaction to add the clients to the decision's attendance list.

- *Ongoing*: When all clients have responded or been timed out, the backend sets the decision state to `'ongoing'`. When the ledger component responds, clients are notified and again respond to affirm their attendance. The backend sets the `hereAtStart` boolean on the ledger state for each responsive client. The backend also starts a timer for each client when their attendance is affirmed, and during its lifetime the clients are free to call `reserve` to vote against the decision. The client should present its user with the same time frame, but keeping a timer at the backend as well protects against malicious or faulty clients delaying the meeting. Note that organizations might still run malicious backends to delay the meeting, as described in Section 3.2.
- *Ended*: When the decided time has passed for all clients, the backend sets the state of the decision to `'ended'` in the same way that it was set to `'ongoing'` previously. Once again, clients are asked to affirm their attendance.

When the decision has been ended, a client can be considered to have participated in the decision without reservation if the `hereAtStart` and `hereAtEnd` booleans are `true` and the `reservation` boolean is `false`.

4.3 The Client

The client software is developed as a standalone Node.js application. The client uses the widely used `node-fetch` and `express` libraries to communicate with the backend over HTTP. When started, the address of the backend service is passed to the client so that it can announce its existence to the backend.

Clients can receive three types of requests from the backend, corresponding to the three different phases of a decision. For each request, the client will automatically respond to affirm its attendance. The consequence of the affirmation depends on the decision state:

- *Decision started.* The client is joined to the decision.
- *Decision ongoing.* The client presents its user with a timer and the option to reserve against the decision.
- *Decision ended.* The decision is over and the outcome can be presented.

5
Experiment

In this chapter, an experiment intended to evaluate the usability of the prototype system is presented. Two metrics are defined to quantify usability, and the results of these metrics at different system scales are presented.

5.1 Methods

Two metrics, *overhead* and *response time*, are defined and measured at different scales in terms of meeting size and network conditions. The overhead metric quantifies the added cost in time of the system to the meeting and the response time metric quantifies the usability of the system from the point of view of a single user. We can call the system *usable*, as defined in Section 1.3, at a given scale and a given overhead cost if the response time is near real-time (defined here as less than three seconds).

5.1.1 Scaling

The prototype system scale is determined by three aspects:

- *Number of orderers.* Increasing the number of orderers is analogous to increasing the number of organizations represented at a meeting, since each organization is expected to run at least one orderer. This increases the time to reach consensus for all transactions.
- *Number of clients.* Increasing the number of clients is analogous to increasing the number of participants of a meeting. This increases the number of transactions for each decision.
- *Network delay between Fabric components.* Increasing the network delay between the components in the distributed ledger increases the time for ordering and validating transactions.

Scaling up the number of orderers and clients is achieved by simply running more instances of the components. All clients and orderers follow the same protocol regardless of the organization to which they belong, so for performance evaluation given a certain consensus algorithm and network delay, the number of clients and orderers is the only relevant configuration factor. In the case of orderers, a change to the system configuration is needed as well since the number of orderers is predetermined. Since all Docker containers in the system run Linux images, network delay can be added for each container by using the `tc` (Traffic Control) program.

5.1.2 The Overhead Metric

The purpose of the overhead metric is to quantify the cost in time that is introduced by using the prototype system. The overhead of a decision introduced by the system is defined as: `total_decision_time - time_to_act`

The total time is defined as the time between when a client initiates a decision (the `startDecision` call to the backend) and when the decision is ended (the `decisionEnded` call from the backend). The time to act is given, but does not matter for this experiment since we will subtract it from the total time to get the overhead.

To get the economic cost added by the system during a meeting, the overhead of a decision can be multiplied with the number of decisions and the number of participants to get the paid man-hours added.

The overhead metric is measured from the point of view of a single client. For this experiment, we will set the time to act at zero, so that the total time equals the overhead for one client. The client starts a timer when initiating a decision, responds to the backend according to the decision process and stops the timer when the decision is over; the resulting time is the overhead at a given scale.

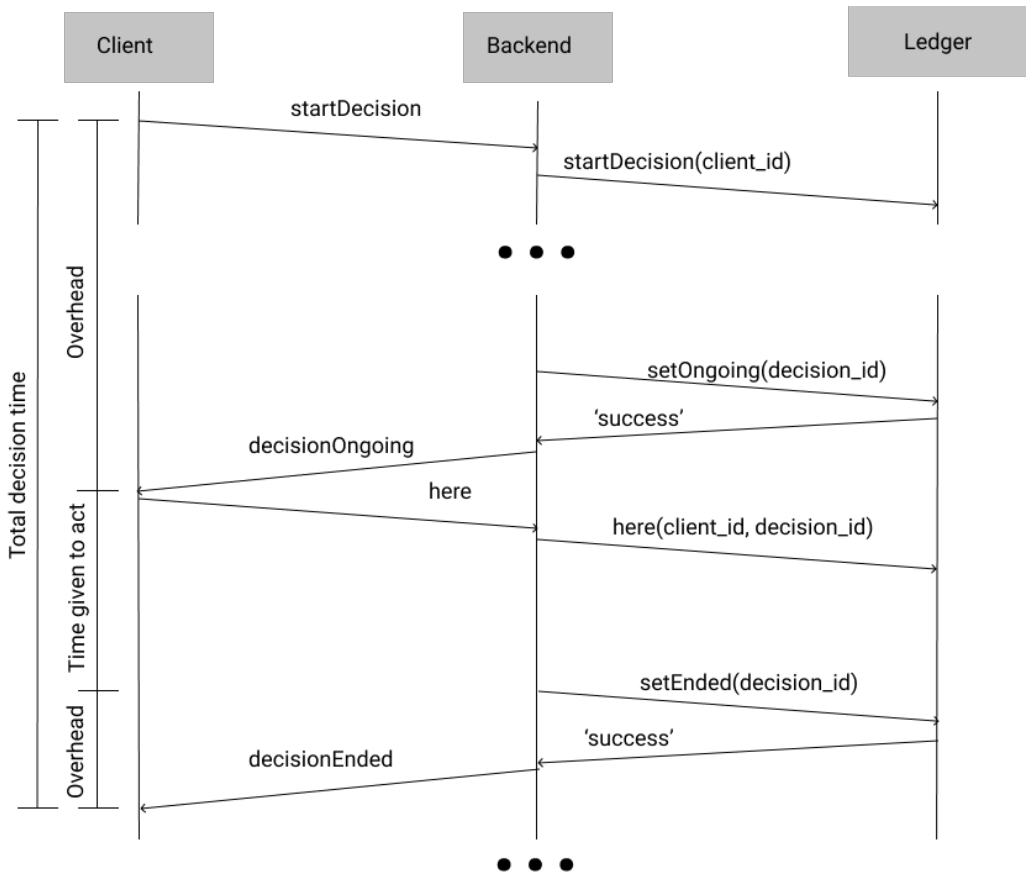


Figure 5.1: A simplified view of the decision process from the view of a single client and the overhead metric.

5.1.3 The Response Time Metric

During the decision process (Figure 4.3), since all organizations are synchronized by relying on the ledger as the source of truth, all transactions resulting from clients asserting their attendance in a given phase and all reserve votes from clients must be resolved before moving on to the next phase. This means that during each decision, some smart contract transactions depend on others rather than relying on an eventual response as could be done if there was no dependency between transactions. We need a *near real-time* response for the system to be usable from the point of view of actual users. The purpose of the *response time* metric is to quantify and evaluate this property.

The response time is defined as the time between a client action resulting in a smart contract transaction and its response during a decision. Some transactions do not have to return anything to the backend, but starting a decision and changing its state have return values after which a response is expected by clients. This response time will be experienced by the users as a delay between an action or event and its effect, and should be near real-time. In our case, after consulting experts at Yolean, we define near real-time as a delay *less than 3 seconds*.

Since the delay in communication between the backend and a client is simply the network delay we set, not the response time of the system, the network delay between the backend and a client is omitted from the results by measuring the response time from the point of view of the backend. The backend measures the response time by starting a timer when a transaction that will result in a response from the ledger is invoked, and stopping it when the response arrives. The relevant transactions are those that have return values; `startDecision`, `setOngoing` and `setEnded` (Figure 4.3).

5.1.4 Procedure

All experiments were run on a 2018 Macbook Pro 15" with the following hardware specifications:

Processor Name:	6-Core Intel Core i7
Processor Speed:	2,2 GHz
Number of Processors:	1
Total Number of Cores:	6
L2 Cache (per Core):	256 KB
L3 Cache:	9 MB
Hyper-Threading Technology:	Enabled
Memory:	32 GB

The prototype system was deployed to a local Kubernetes cluster, itself contained in a Docker container running on the machine using the tool *Kubernetes-in-Docker* (kind). Docker was allowed to use 16 GB RAM and 6 CPUs.

To make the prototype system experimental deployment as simple as possible while producing accurate results, it consisted of a single organizations running:

- A single backend

- A single Fabric peer
- One of the following two ordering services:
  - A single BFT-SMaRt frontend and a number of BFT-SMaRt ordering nodes
  - A single Raft ordering node
- A number of clients

In Fabric, following the Execute-Order-Validate paradigm (see Section 2.2.2), the only part that is significantly affected by increasing the number of replicas is the ordering. The execution and read-write conflict validation does not depend on any other replica and the signature validation is done in parallel. Regardless, the execution and validation are logged in the peer container and was observed to only take a few milliseconds when running the experiments. The time it takes to order events is vastly longer than both execution and validation, so increasing the number of ordering nodes has the same effect on the experiments as increasing the number of organizations with one ordering node each.

The tests were conducted by setting up Fabric with a certain number of ordering nodes and network delay. The overhead was measured by iteratively initiating the decision process with an increasing number of clients and the response time was measured by invoking a single transaction. Since external circumstances like CPU temperature and thermal throttling can affect the result, the tests were run multiple times and the final results are the averages of those runs. Each overhead test was run three times and each response time test was run ten times.

The scales used were:

- `BFT Ordering nodes: 4, 7, 10`
- `Network delay: 0ms, 100ms, 200ms`
- `Clients: 20, 40, 60, 80, 100`

Since `3m+1` ordering nodes solves BFT consensus in the presence of `m` faulty nodes (see Section 2.3.2), the number of ordering nodes corresponds to allowing up to 1, 2 and 3 faulty nodes. The network delay of 0ms corresponds to running all nodes in the same physical network while the delays of 100ms and 200ms are meant to more accurately reflect the network delay between physically separate networks. The number of clients are tied to the aim (section 1.3), where 20 clients would be the lowest practical size and 100 clients would be a large meeting.

5.2 Results

In this section, the results of both metrics at the specified scales are presented. The different scales in terms of BFT ordering nodes are presented in comparison to a *CFT baseline*; results of the same system with the CFT ordering service, to quantify the response time and overhead further added to the system by the BFT ordering service.

5.2.1 Overhead

The overhead of the prototype system using the BFT ordering service is consistently higher than the CFT baseline. Increasing the number of clients increases the over-

head in a similar way for both ordering services, but an increased network delay affects the BFT ordering service significantly more.

Overhead in seconds, 0ms network delay				
#clients	CFT baseline	4 BFT nodes	7 BFT nodes	10 BFT nodes
20	14	22	23	25
40	23	30	32	40
60	32	40	45	55
80	45	52	63	72
100	64	70	86	94

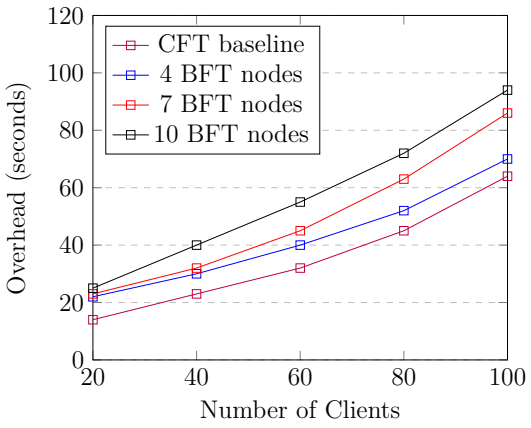
Table 5.1: The overhead in seconds with no network delay added. Each result is the average of three runs.

Overhead in seconds, 100ms network delay				
#clients	CFT baseline	4 BFT nodes	7 BFT nodes	10 BFT nodes
20	15	24	26	27
40	25	32	38	41
60	34	42	53	57
80	46	55	66	73
100	65	75	89	96

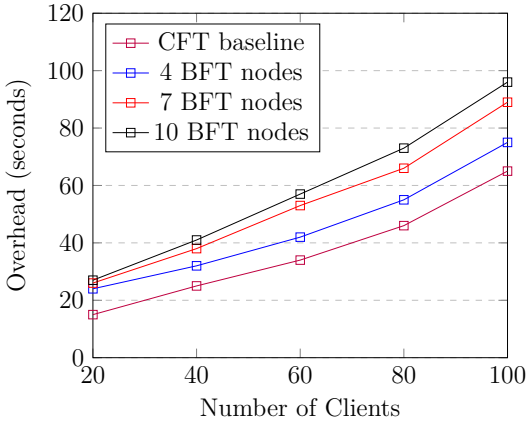
Table 5.2: The overhead in seconds with 100ms network delay added. Each result is the average of three runs.

Overhead in seconds, 200ms network delay				
#clients	CFT baseline	4 BFT nodes	7 BFT nodes	10 BFT nodes
20	18	26	28	31
40	26	36	42	44
60	35	50	62	65
80	47	65	83	87
100	65	86	106	110

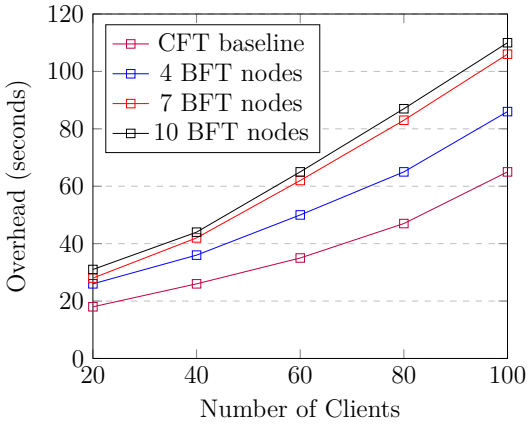
Table 5.3: The overhead in seconds with 200ms network delay added. Each result is the average of three runs.



(a) 0ms network delay



(b) 100ms network delay



(c) 200ms network delay

Figure 5.2: The overhead in seconds of 4, 7, and 10 BFT nodes at 0ms, 100ms and 200ms of added network delay.

5.2.2 Response Time

The response time using the BFT orderer is significantly higher than the baseline, and increases faster when adding network delay. Adding more BFT nodes does not affect the response time significantly.

Response time in seconds				
Network delay	CFT baseline	4 BFT nodes	7 BFT nodes	10 BFT nodes
0ms	1.2	2.2	2.3	2.3
100ms	1.5	2.7	2.8	2.8
200ms	1.8	3.3	3.4	3.4

Table 5.4: The response time in seconds. Each result is the average of ten runs.

6
Discussion

In this chapter, the results of the experiment are analyzed and discussed in relation to the aim of the thesis, and future work is suggested. The thesis is also discussed in the context of ethics and sustainability.

6.1 Results

Overall, the results show that the prototype system adds a significant overhead to meetings at larger scales, especially in terms of the number of clients. With low levels of network delay between the organizations, the response time can be considered near real-time, but the overhead would introduce a very high cost to the meeting. For example, with 4 organizations and 40 attendants present and a network delay of 100ms, the overhead is 32 seconds (see Figure 5.2) and *21 working minutes* $(32 * 40/60)$ are added for every decision during the meeting.

6.1.1 Overhead

As expected, due to the high complexity of BFT consensus, the overhead added by the BFT ordering service is significant.

Increasing the number of clients increases the number of attendance events correspondingly, but the transactions resulting from attendance assertions do not depend on each other so an increase in overhead by a factor of less than one was expected. However, the BFT ordering becomes much slower when concurrent events are introduced and the chance that nodes disagree on the order increases, which explains the significant increase in overhead at larger scales. For the same reason, increasing the number of replicas had a greater significance with more clients in attendance.

When increasing the network delay, the overhead of the CFT baseline only increased slightly, and the increase in overhead from increasing the number of clients was not affected much. As can be seen in Tables 5.1, 5.2 and 5.3, the overhead for the CFT baseline at 100 clients was within one second for all three levels of network delay. In contrast, due to the increased message round trips of BFT consensus, the overhead for the BFT ordering service at 100 clients is significantly increased by network delay.

At smaller scales, corresponding to a conference room-sized meeting with a small number of clients and only a few organizations in attendance, the overhead could be considered acceptable. However, at larger scales than that, the overhead cost is unreasonably high.

6.1.2 Response Time

The response time remained sufficiently low for all cases except for when adding 200ms of network delay to the system using the BFT orderer, and in that case still less than half a second over our specified near real-time limit of 3 seconds. Looking at the results, a few observations can be made (see Figure 5.4):

- Using the BFT ordering service increased the response time by at least a second compared to the baseline.
- Increasing the number of BFT nodes does not affect the response time significantly
- Increasing the network delay increases the response time approximately by a factor of 3 in the baseline case and a factor of 5 in the BFT case.

The greater impact of network delay on the BFT ordering service can be explained by the higher amount of message round trips required. This also explains why adding more BFT nodes does not affect the response time significantly; the same amount of round trips are still required, and all ordering nodes will probably agree on the order of a single transaction.

6.1.3 Factors

In *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform* by João Sousa et al. [31], the performance of the ordering service based on BFT-SMaRt is evaluated and notably, block size (the number of transactions that are grouped into a block to be appended to the blockchain) and envelope size (the size of the transactions) play a big role when it comes to throughput. It is concluded that for large envelopes, ordering becomes the dominant factor while for small envelopes, the CPU-bound work related to creating blocks of transactions becomes dominant. When ordering is the dominant factor, the number of ordering nodes significantly affects the result. When block creation is the dominant factor, all transactions are ordered faster than blocks are created, so grouping more transactions into a single block greatly increases performance while the number of ordering nodes matters less.

Due to the following observations, the bottleneck in our experiments seems to be the ordering of events:

- Increasing the number of ordering nodes significantly increased the overhead;
- Increasing the network delay significantly increased the overhead;
- Increasing the block size did not affect the overhead.

The result of increasing the network delay and number of orderers in our case can be seen in Figure 5.2. Some experimentation was done with the block size but it did not affect the results in any significant way. This might be due to a relatively large envelope size (our envelope size was not explicitly evaluated, but the signature generation time was only found to become dominant at envelope sizes of 40 and 200 bytes, compared to larger envelopes of 1 and 4 kbytes [31]), but might also be affected by the fact that, rather than a constant stream of transactions, our system produced short bursts of transactions during certain times of the decision process, corresponding to the number of clients. It is noted that signatures are generated drastically slower when the CPU is shared by both signature generation and BFT-

SMaRt [31]. The impact of this might be mitigated by the fact that only a limited number of signatures are generated during a short period of time at the start of a burst of transactions, followed by longer periods with no new transactions.

Ordering time per message: As can be seen in Table 5.1, the difference between 7 and 10 BFT nodes increases from 2 to 8 seconds when doubling the number of clients from 20 to 40. After that, a further increase in the number of clients does not seem to significantly increase the difference between 7 and 10 BFT nodes further. When looking at the five-fold increase of 20 to 100 clients and comparing the 0ms network delay results (Figure 5.1 and the 200ms network delay results (Figure 5.3, we see that the increase is roughly four-fold in the 0ms case, and three-fold in the 200ms case.

This suggests that, when a batch of messages are received by an ordering node, the later messages are processed faster than the earlier. The cause of this is not investigated or made clear by the results of this thesis, but since the signature generation time would be constant or increase under a high load on the node, it further suggests that the limiting factor is the ordering of messages.

6.1.4 Possible Improvements

The major issue with the prototype system is the high overhead. However, there are several ways in which the overhead of the system could be reduced.

6.1.4.1 Reduce the Number of Participating Clients

The vast majority of transactions are attendance assertions from the clients, and these transactions are also all invoked simultaneously for all clients, so relaxing the requirements here could drastically decrease the number of concurrent transactions.

One way to do this would be to only require attendance from a subset of clients for each organization before changing the decision state and moving on with the process. This would change the guarantees of the system so that instead of ensuring that *all* clients have taken part of all information before moving on, we ensure that *some* clients in each organization are up to date. The other clients could be expected to answer shortly after if the system replica of the organization is working correctly and if they do not, we can disregard them, note that we do not know exactly which information they took part of before leaving, but still know that some clients in the same organization received all information.

6.1.4.2 Reduce the Number of Concurrent Transactions

Concurrent transactions could also be reduced by reducing the dependency between decision state changes and attendance. Instead of asserting the attendance of clients after each phase, we could assert attendance gradually during the decision process and require each client to respond with its current view of the meeting state. If some client falls behind in this case, we know the last asserted state and can assume that everything after that point might not have been seen.

The decision process would work fine under this decision process, but the amount of information available in real-time for other clients would be reduced. Instead

of being able to deliver an up-to-date view of the current state to any client at any time, the view would reflect, and decisions would be started with, the clients assumed to still be in attendance. This changes the adversary model and reduces the guarantees of the system; rather than letting the system determine what each client knows, clients could lie about their current view.

6.1.4.3 Use a Crash Fault Tolerant Consensus Algorithm

Finally, the BFT ordering service could be abandoned in favor of a CFT ordering service like Raft. As seen in the results, this greatly decreases the overhead and response time of the system, especially at larger scales. Due to the strong leader node of Raft (see Section 2.3.1.2), the difference in performance would be especially significant without the presence of faults.

Obviously, this also reduces the guarantees of the ordering service, but all other advantages in security and trust of using a permissioned distributed ledger rather than some centralized solution are still present. In the presence of arbitrary behavior from some ordering nodes, transactions could be disregarded and the order of transactions could be determined by malicious actors, but the validity of transactions and the immutability of the log are still ensured by the system.

6.2 Ethics and Sustainability

The ability to enable trust between parties in different contexts regardless of physical proximity can have several societal benefits. Remote meetings leads to reduced travel which avoids emissions and disease transmission, and have the potential of reducing the economic costs. However, as has been shown in this thesis, a system enabling this can also come with a large cost in terms of time an energy, and the advantages of reduced travel must outweigh these costs.

Distributed ledgers, and in particular cryptocurrency blockchains, are generally known to have an extremely high power consumption due to their proof-of-work consensus methods. In *The Energy Consumption of Blockchain Technology: Beyond Myth* by Johannes Sedlmer et al. [29], the energy consumption of some proof-of-work blockchains is analyzed and compared to other systems.

Some interesting observations are made. Specifically, the energy consumption of proof-of-work blockchains do not directly depend on the rate of transactions. The energy consumption of a system like Bitcoin does not necessarily increase when it is used more, but rather when it is mined more, and the incentive to mine Bitcoin blocks is not to increase the transaction rate but to increase the chance of finding a block. It is the voting weight of a single node that correlates with processing power and energy consumption, not the rate of transactions, and there are ways of increasing the transaction rate without increasing the energy consumption. One way is to increase the block size, meaning the number of transactions that are grouped into a block. This is the idea behind Bitcoin cash, a widely used fork of bitcoin that increased the block size from 4MB to 8MB.

Permissioned blockchains that do not rely on proof-of-work are discussed in general, and both Hyperledger Fabric and the practical Byzantine fault tolerance

algorithm are mentioned which is especially relevant to this thesis. Even though the energy consumption of permissioned distributed ledgers increases due to higher rates of transactions in a much more direct way, due to the high message complexity of their consensus algorithms, the energy consumption is orders of magnitude lower than their permissionless proof-of-work counterparts [29].

As mentioned in Section 1.1, the permissionless blockchain Ethereum popularized the concept of smart contracts and the use of distributed ledgers to solve more general problems than cryptocurrency transactions. If applications that do not necessarily depend on the permissionless nature of such systems can be implemented in permissioned systems instead, the energy consumption of those applications would be drastically reduced.

Ethics: The ethical concerns most often raised about blockchain technology has to do with its shared nature. All positive aspects in terms of privacy and decentralization have the disadvantage of reducing the possibility of desirable oversight and control. Some issues that permissionless systems have, like the possibility of anonymous illegal trade, are not relevant in our case since a permissioned system is made up of known and authenticated nodes. However, the lack of a central authority in a permissioned system can still lead to problems due to its immutability.

For example, a poorly designed or insecure system can leak secret or personal data. If that data ends up on a distributed ledger used by that system, it is very hard to remove that data. In our case, some part of the consortium might accidentally reveal sensitive customer information or information that give other parts of the consortium an advantage in negotiations. After being immutably logged in each ledger replica, there are several copies of this data that cannot be removed without the cooperation of every other part of the consortium. Using technologies like blockchain is in no way a guarantee that a system is secure in general, and if other parts of the system are poorly designed, it can even reduce the security of the system in many ways.

6.3 Future Work: Byzantine Fault Tolerant Consensus in Permissioned Distributed Ledgers

Even though the possible contexts in which distributed ledgers can be used have been greatly increased by the introduction of permissioned systems and smart contracts, the technology still has some major drawbacks. In particular, throughput remains an issue for distributed ledgers, even though the throughput of a permissioned distributed ledger like Fabric is much higher than a public permissionless ledger like Ethereum.

It is very possible that some other BFT algorithm implementations would perform significantly better under our specific circumstances. In *State Machine Replication for the Masses with BFT-SMART* by João Sousa et al. [3], it is noted that there are not many practical and production-ready implementations of BFT algorithms and in *In Search of an Understandable Consensus Algorithm* by Diego Ongaro et al. [23], the same point is made about the crash fault tolerant Paxos and used as a motivation for Raft. When it comes to implementations for permissioned distributed

ledgers, implementations are even more scarce. Hyperledger Fabric was intended to get a BFT ordering service in release 1.0 [28], and will eventually get an official BFT order based on the same decentralized design as Raft [28, 18].

The BFT-SMaRt-based ordering service [31] used in this work is as of now the only BFT option for Fabric, and the need to rely on frontends and a separate ordering service cluster (see Section 4.2.3) is a major drawback. Getting it running reliably was also a challenge, especially with a higher number of ordering nodes. Often, one or more ordering nodes would crash at startup and with certain combinations of ordering nodes and block sizes, blocks would sometimes stop being created. Both these issues would cause the system to stop processing transactions and require a restart.

Before a system based on Fabric that provides Byzantine fault tolerance can realistically be used in production, more work is needed in this area.

7
Related Work

This thesis builds a lot on top of existing research relating to distributed ledgers and consensus methods. The concept of blockchains is still relatively new and the possible uses of different variants of blockchain technology is constantly evaluated. Permissioned distributed ledgers is a new technology where the established technology of BFT consensus found new uses. Miguel Castro and Barbra Liskov presented pBFT in 1999 [7], but, as stated by João Sousa et al. [3], in 2014 there were still not many practical implementations used in actual systems. Now, when talking about distributed consensus in general and BFT consensus in particular, it is often in the context of blockchains, and new consensus methods building on top of the established work are often a direct result of the needs of blockchains. An example of this is *The Istanbul BFT Consensus Algorithm* by Henrique Moniz [20], which was publsihed in 2020 and is the consensus algorithm used in the *Quorum* blockchain.

One example of a work that solves a similar problem to the one described in this work is *A Smart Contract for Boardroom Voting with Maximum Voter Privacy* by Patrick McCorry et al [19]. The work aims to provide a boardroom voting system based on the blockchain Ethereum, which guarantees voter privacy and calculates the tally without relying on any trusted authority. This is very similar to our work in terms of the problem and solution; voting during a meeting is solved by using distributed ledgers and a smart contract. However, this work differs in terms of both aim, design and implementation in important ways:

- *Use case.* Even though trusted boardroom voting seems very similar to a trusted decision log, the intended use is fundamentally different. Privacy is not a concern for the system presented in this thesis. Rather, we use a permissioned distributed ledger where the identities of all participants are ensured and clear to all. We also do not care about vote tallying or anything similar. Rather, the model of decision making for this work is *silent accept*, which means that rather than counting votes on decisions, the participants and whether or not they oppose a given decision is logged. This implies that consensus has to be reached much more often than just once for each decision and a greater throughput is needed, which leads us to the next point.
- *Scale.* The referenced work states that they chose the scale of boardrooms or a maximum of approximately 40 voters, rather than national scale voting, because of the scaling limitations of Bitcoin and Ethereum. Since the intended use of the work in this thesis is meetings within a consortium of organizations, participants have to be authenticated beforehand. This fact removes the need to rely on the proof of work-based consensus algorithms of systems mainly intended for cryptocurrencies. Rather, a consensus method suitable for the

specific needs of the system is used, which increases the throughput of the system significantly.

The prototype system developed in this work is capable of, but also requires, a much higher throughput of transactions than is possible in a public permissionless distributed ledger like Ethereum. For problems similar to the one outlined in the referenced work [19], the use of permissioned distributed ledgers might be able to greatly increase the feasible scale.

Performance Evaluations of Hyperledger Fabric: There are a number of papers that evaluate the performance of Hyperledger Fabric under different circumstances. The performance evaluation of this work is similar in some ways, but differs in the fact that they evaluate performance from a generic point of view, rather than the way in which performance is evaluated for a specific purpose in this work.

One example is *Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform* by Parth Thakkar et al. [32], which among other things evaluates transaction *throughput* and *latency* at different transaction arrival rates, which roughly corresponds to *overhead* and *response time* for different numbers of clients in this work. However, they only use a single ordering node backed by a Kafka cluster (the default ordering service before the Raft-based ordering service was released in version 1.4.1), and find that ordering does not become a bottleneck until an arrival rate of around 140K transactions per second, which is much later than in our case where the ordering service is more complex and ordering very quickly becomes the dominant factor. For future work they mention, among other things, comparing different consensus algorithms at different scales, introducing network delay between nodes and testing more realistic workloads rather than a constant transaction arrival rate, which are all factors taken into account in the experiment of this work.

Another example is *Impact of network delays on Hyperledger Fabric* by Thanh Son Lam Nguyen et al. [22], which also uses the old Kafka-based ordering service. Their methodology consists of hosting nodes in two separate clouds, adding a delay between them in the same way in which delays are added between nodes in this work, and observing the time between when a block is added in one cloud compared to the other. They observe the time difference between blocks to increase in a linear fashion when increasing the delay, which is expected unless conflicts or faults occur that cause the number of messages required to reach consensus to increase and is consistent with the results of the experiment of this work.

8

Conclusion

In this thesis, a prototype system of a secure, trustworthy and usable decision log, based on the permissioned distributed ledger *Hyperledger Fabric* and intended to be used in large-scale meetings within consortiums of organizations was developed and evaluated.

Security was provided by Fabric implementing a blockchain to log all smart contract transactions, making the log immutable and tamper proof. *Trust* was provided by the system architecture; each organization runs their own software stack, including a distributed ledger replica, and all information between organizations is exchanged through the ledger replicas reaching consensus about proposed transactions. Additionally, two different methods of consensus were implemented; one providing *crash fault tolerance* and the other providing the stronger *Byzantine fault tolerance*. Byzantine fault tolerance provides a higher level of trust by tolerating arbitrary behavior from some replicas, but is slower and more complex than its crash fault tolerant counterpart.

The *usability* of the system was evaluated in a series of experiments. Two metrics intended to quantify usability were defined; *overhead* and *response time*. The overhead quantifies the amount of time added by the system to the meeting as a whole and the response time quantifies the delay between user actions and the corresponding response from the system.

The response time was found to be sufficiently low in all cases. However, the overhead was found to be very high, especially when the number of participating clients reached close to a hundred; reflecting a large meeting in the construction industry, which makes the system very expensive to use. The Byzantine fault tolerant consensus method further increased the overhead of the system. To decrease the overhead to reasonable levels, a drastic decrease of the number of concurrent transactions or a drastic increase in transaction throughput is required.

Bibliography

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric. *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*, 2018.

[2] Adam Back. Hashcash - a denial of service counter-measure. 09 2002.

[3] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1):70–93, 2016.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[8] Usman W Chohan. The limits to blockchain? scaling vs. decentralization. *SSRN Electronic Journal*, 01 2019.

[9] Wei Dai. b-money. `http://www.weidai.com/bmoney.txt`, 1998. Accessed: 2020-07-08.

[10] Hyperledger fabric samples. `https://github.com/hyperledger/fabric-samples`. Accessed: 2020-06-29.

[11] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[12] Gateway. https://hyperledger-fabric.readthedocs.io/en/release-2.0/developapps/gateway.html. Accessed: 2020-07-17.

[13] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. *arXiv preprint arXiv:1911.09208*, 2019.

[14] About Hyperledger. https://www.hyperledger.org/about. Accessed: 2020-06-25.

[15] KIDSAM: Knowledge and information-sharing in digital collaborative projects. https://www.chalmers.se/en/projects/Pages/KIDSAMQ-Knowledge--and-information-sharing-in-digital.aspx. Accessed: 2019-12-15.

[16] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[17] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[18] Arnaud Le Hors. Demystifying Hyperledger Fabric ordering and decentralization. https://developer.ibm.com/articles/blockchain-hyperledger-fabric-ordering-decentralization/. Accessed: 2020-07-12.

[19] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 357–375, Cham, 2017. Springer International Publishing.

[20] Henrique Moniz. The Istanbul BFT Consensus Algorithm. *arXiv e-prints*, page arXiv:2002.03613, February 2020.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[22] Thanh Son Lam Nguyen, Guillaume Jourjon, Maria Potop-Butucaru, and Kim Loan Thai. Impact of network delays on Hyperledger Fabric. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 222–227. IEEE, 2019.

[23] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[24] The ordering service. https://hyperledger-fabric.readthedocs.io/en/release-2.0/orderer/ordering_service.html. Accessed: 2020-07-17.

[25] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[26] Hyperledger Fabric meets Kubernetes. `https://github.com/hyfen-nl/PIVT`. Accessed: 2020-09-30.

[27] The Raft consensus algorithm. `https://raft.github.io/`. Accessed: 2020-07-15.

[28] Regarding Byzantine fault tolerance in Hyperleder Fabric. `https://lists.hyperledger.org/g/fabric/topic/regarding_byzantine_fault/17549966?p=,,,20,0,0,0::recentpostdate%2Fsticky,,,20,2,240,17549966`. Accessed: 2020-07-12.

[29] Johannes Sedlmeir, Hans Ulrich Buhl, Gilbert Fridgen, and Robert Keller. The energy consumption of blockchain technology: beyond myth. *Business & Information Systems Engineering*, pages 1–10, 2020.

[30] Smart contracts and chaincode. `https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html`. Accessed: 2020-06-29.

[31] Joao Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[32] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing Hyperledger Fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276. IEEE, 2018.

[33] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.