

Paxos Blockchain

A private blockchain simulation based on the Paxos consensus algorithm

Date: June 18, 2018

Version: 1.2

Author

Name: Florian van Herk

Student number: 0898132

Organisation

Centric B.V. Gouda

Antwerpseweg 8

2803 PB Gouda

The Netherlands

University

Rotterdam University of Applied Sciences

Wijnhaven 107

3011 WN Rotterdam

The Netherlands

Company supervisor

Name: Prof. Dr. Ben van Lier

Email: ben.van.liet@centric.eu

First university supervisor

Name: Ahmad Omar

Email: omara.hr.nl

Second university supervisor

Name: John Grobber

Email: j.grobber@hr.nl

Preface and acknowledgments

Going into this assignment, I had no idea what to expect. After searching for weeks - or months, even - I still had no luck with finding a final internship for graduation. Luckily, Centric saw an opportunity in me (at one of the last moments I could even obtain an internship, mind you), and accepted me into the company. Aside, from that, the assignment turned out to be something which I found to be both fun, and something which would be a great asset to my knowledge: (1) the rising technology, namely blockchain, (2) networking, (3) distributed systems, and (4) consensus algorithms.

Therefore, I would like to express my gratitude towards my company supervisor Ben van Lier, for providing me with this opportunity of researching the topic of blockchain and consensus algorithms, and for his guidance throughout the research. Aside from that, I would also like to thank my fellow interns Denny Koemans and Dirk-Pieter Jens, for sharing their knowledge on blockchain, keeping up the spirit, and especially for the company they provided me throughout my research.

From an academic perspective, I would like to thank my teachers/coaches from the university, Ahmad Omar and Jon Grobber, for their advice on approaching my research. Also, a special thanks to my classmate Corné Verhoog, for sharing his excellent knowledge and ideas on computer networking. With his help I was able to successfully build the foundation of the blockchain simulation - the blockchain network layer.

Lastly, I would like to thank my family, my school friends, my band friends, my best friend Casper and all of my other friends for their love, encouragement and support. With their help I was able to finish this research successfully.

Abstract

Blockchain is a revolutionary technology, which allows multiple parties to share information without the use of a Trusted Third Party (TTP). Most conventional blockchains are public blockchains, which use an external method for validating transactions made on the blockchain. On the other hand, Centric wants to implement a private blockchain, required for a project between multiple different companies. The data shared on this private blockchain is confidential—therefore, an internal method of validating these transactions is required. The Paxos consensus algorithm is possibly a solution to this problem, since it appears to contain all the required characteristics for reaching consensus on values written to a distributed ledger.

The goal of this research is to understand the value of the Paxos consensus algorithm within the context of a private blockchain. The main question of this research is: “*What value does the Paxos algorithm have for reaching consensus in the context of a blockchain?*”. Important sub questions are: (1) How does Paxos ensure the required qualities of blockchain, such as immutability of the data and reaching consensus? (2) Is Paxos Byzantine Fault Tolerant?

To answer these questions, a case study was conducted on the article *The Part-Time Parliament*, which contains an in-depth description of the Paxos algorithm. Based on the description, an implementation of the Paxos algorithm was written in .NET core. Besides, multiple tests have been written to understand the qualities and shortcomings of the Paxos algorithm.

The results show that the *Synod protocol* described in *The Part-Time Parliament* is a well functioning, effective algorithm for reaching consensus on solely one value – so not very valuable for a blockchain. On the other hand, the *Multi-decree protocol* – which is derived from the *Synod Protocol* is able to reach consensus on a subsequent series of values. Only one aspect of the made implementation keeps it from being fully *Fault Tolerant*. The *Multi-decree protocol* assumes a static number of online nodes, but perhaps an error has been made in the implementation, due to the vague specification of the *Multi-decree protocol*. Furthermore, Paxos contains no *Byzantine Fault Tolerance* properties on its own, and the possible given solution in *The Part-Time Parliament* remains unclear. Lastly, Paxos assumes immutability is preconfigured, since no explanation can be found on how to achieve this *immutability*. Setting triggers on database level prevent users from deleting and updating decrees, however nothing is stopping users from inserting new data.

From this we can conclude that the *Multi-decree protocol* described in *The Part-Time Parliament* is a valuable consensus algorithm to be used in a private blockchain, as long as further research is done on (1) how to deal with its single shortcoming found in the implementation, (2) if a method can be found to reinforce Paxos with Byzantine Fault Tolerance, and (3) if a method has been found for preventing data manipulation by hand. Finally, applying a state-machine could be done for realizing a more realistic business case.

Samenvatting

Blockchain is een revolutionaire technologie, dat verschillende partijen toestaat om met elkaar informatie te delen zonder dat hier een vertrouwde derde partij bij komt kijken. De gewoonlijke blockchains zijn veelal public blockchains—deze zijn toegankelijk voor elk persoon, en gebruiken externe methodes voor het valideren van de transacties. Centric wilt echter gebruik maken van een private blockchain; verschillende bedrijven willen met elkaar samenwerken door data met elkaar te delen. Aangezien betrouwbare informatie gedeeld zal worden op deze blockchain, willen ze geen public blockchain, en is er dus een manier nodig die het mogelijk maakt om intern de transacties te valideren. Het Paxos consensusalgoritme lijkt op de oplossing voor dit probleem, omdat het alle eigenschappen lijkt te hebben voor interne validatie van data van een gedistribueerde ledger.

Het doel van dit onderzoek is om te begrijpen wat de meerwaarde is van het Paxos consensusalgoritme binnen de context van een private blockchain. De hoofdvraag van dit onderzoek luidt: “*Welke toegevoegde waarde heeft het Paxos algoritme voor het bereiken van consensus binnen de context van blockchain?*”. Belangrijke deelvragen zijn: (1) Hoe verzekert Paxos de vereiste kwaliteiten van blockchain, zoals onveranderlijkheid van de data, en het bereiken van consensus? (2) Is Paxos Byzantine Fault Tolerant?

Om een antwoord te krijgen op deze vragen, is een case study toegepast op basis van het artikel *The Part-Time Parliament*. Dit artikel bevat een gedetailleerde omschrijving van het Paxos algoritme. Op basis van het algoritme is een implementatie van het Paxos algoritme gemaakt in .NET core. Daarnaast, om de gemaakte implementatie beter te begrijpen en te onderbouwen zijn verschillende tests geschreven en uitgevoerd.

Uit het resultaat is gebleken dat het *Synod protocol* omschreven in *The Part-Time Parliament* een goed functionerende, effectieve algoritme is om tot consensus te komen in een onvoorspelbare, asynchrone omgeving—echter maar voor één enkele waarde. Het *Multi-decree protocol* (ontstaan vanuit het *Synod protocol*) daarentegen is in staat om tot consensus te komen over een reeks van waarden. Echter is er één pijnpunt dat ervoor zorgt dat de implementatie niet volledig *Fault Tolerant* is. Het *Multi-decree protocol* gaat uit van een vaste hoeveelheid deelnemers, alhoewel zou het kunnen zijn dat de omschrijving verkeerd geïnterpreteerd is; het *Multi-decree protocol* is minder nauwkeurig omschreven in het artikel dan het *Synod protocol*. Daarnaast bevat Paxos geen eigenschappen dat het *Byzantine Fault Tolerant* maakt, en de gegeven oplossing is wederom onnauwkeurig en onduidelijk omschreven. Ten derde, Paxos neemt aan dat de onveranderlijkheid (immutability) van de data ergens anders geconfigureerd wordt, aangezien het artikel geen omschrijving bevat hoe het gedaan moet worden. Een voorbeeld van hoe dit bereikt kan worden, is door middel van triggers op database niveau. Deze triggers kunnen gebruikers weerhouden om geschreven data te verwijderen of aan te passen. Echter is er niets dat de gebruiker tegenhoudt om nieuwe data met de hand in te voeren.

Op basis hiervan kunnen we concluderen dat het *Multi-decree protocol* omschreven in *The Part-Time Parliament* wel degelijk een waardevolle consensusalgoritme is voor een private blockchain, zolang in de toekomst onderzoek gedaan wordt naar (1) hoe omgegaan kan worden met de tekortkoming van de implementatie van het *Multi-decree protocol*, (2) of een methode toegepast kan worden waardoor Paxos wel *Byzantine Fault Tolerant* is, en (3) of er een methode gevonden kan worden waardoor gebruikers de data niet kunnen manipuleren buiten de applicatie. Uiteraard heeft het ook meerwaarde om te kijken naar de implementatie van een state-machine voor Paxos, voor het realiseren van een werkelijke business case.

Contents

1	Introduction	1
1.1	Reason of research	1
1.2	Research value & stakeholders	1
1.2.1	Research value	1
1.2.2	Stakeholders	2
1.3	Company, workplace & tasks	2
1.3.1	Company	2
1.3.2	Workplace & tasks	2
1.4	Goal	3
1.5	Problem statement	3
1.5.1	Paxos	3
1.5.2	Network layer	4
2	Methods	5
2.1	Research method	5
2.2	Gathering information	5
2.3	Quality requirements	5
2.4	Research validation	5
2.5	Sources validation	6
2.6	Project method	6
3	Examination of The Part-Time Parliament	7
3.1	The author	7
3.2	Story behind the article	7
3.3	The Paxos parliament	8
3.3.1	Requirements	8
3.3.2	Comparison with distributed databases	9
3.4	Choosing a decree	10
3.5	Versions of the protocol	11
3.5.1	Preliminary protocol	12
3.5.2	Basic protocol	13
3.5.3	Synod Protocol	14
3.5.4	Multi-decree protocol	15
3.5.5	Byzantine Failures	16
3.6	Recap	17
4	Paxos blockchain simulation development	18
4.1	Introduction	18
4.2	Developing the network layer	18
4.2.1	Requirements	18
4.2.2	Choosing the right transport layer	20
4.2.3	Gathering connection information	21
4.2.4	Sending messages between nodes	22
4.2.5	Reinforcing UDP	24

4.3	Developing Paxos	24
4.3.1	Node roles	24
4.3.2	Ledgers	25
4.3.3	Implementing the Synod protocol	27
4.3.4	Implementing the Multi-Decree protocol	43
4.4	Recap	56
5	Conclusion	57
6	Discussion	58
7	Reflection	60
7.1	Teacher's concerns	60
7.2	Understanding the Part-Time Parliament	60
7.3	Development environment	60
7.4	Illness	60
7.5	Validation	61
7.6	Ambition	61
7.7	Overall thoughts	61
Appendix A	Synod protocol class diagram	62
Appendix B	Test results	64
B.1	General test results	64
B.1.1	Immutability	64
B.1.2	Testing the network	64
B.1.3	Excluding a node	65
B.1.4	Corrupt message	65
B.2	Synod protocol test results	66
B.2.1	No simple majority formed	66
B.2.2	Simple majority formed	67
B.2.3	President goes offline before sending BeginBallot message	68
B.2.4	President goes offline before sending Success message	69
B.2.5	Non-president goes offline before sending LastVote message	70
B.2.6	Non-president goes offline before sending Voted message	71
B.2.7	Non-president goes offline before receiving Success message	72
B.2.8	Multiple proposals at the same time	73
B.3	Multi-decree protocol test results	74
B.3.1	Proposing a series of decrees from different nodes	74
B.3.2	Learning decrees - new president	77
B.3.3	Learning decrees - non-president	78
B.3.4	Filling gaps	79
B.3.5	Overwriting olive decrees	82
B.3.6	President shuts down after initiating ballot	84
B.3.7	Multiple proposals at the same time	85
B.3.8	Taking leadership	87
B.3.9	Node joins the quorum after president received all LastVote messages	88
Appendix C	Witness statements	90
Literature		94

Terminology

asynchronous

An asynchronous event or process is able to operate concurrently to other events and processes.

blockchain

A network of computers, which all keep their own copy of a database: a shared ledger. Data on a blockchain is immutable.

byzantine generals problem

the issue where at least one node of a network application is faulty - either intentionally or unintentionally - and sends corrupt/imperfect/malicious information to other nodes.

consensus algorithm

One of the fundamental algorithms of a blockchain, which is used to reach agreement over what data gets added to the shared ledger. Consensus algorithms are usually fault tolerant.

distributed ledger

A database, shared across a network. Every node of the network contains a copy of this database.

node

A node is a computer which is a participant of a computer network. This can be any kind of device with internet access, such as a desktop computer, laptop, mobile phone, smart television, etc.

paxos

An algorithm developed by Leslie Lamport, used to reach consensus among many computers on a distributed system.

peer

A node becomes a peer when they establish a connection with another node. Then these two nodes can be considered to be 'peers'.

private blockchain

A blockchain which is not open to the general public. Instead, one or more organisations control who can participate in the blockchain.

List of acronyms

CRUD Create, Read, Update, Delete.

CSV Comma Separated File.

ORM Object-relational mapping.

P2P peer-to-peer.

RUDP Reliable User Datagram Protocol.

SQL Structured Query Language.

TCP Transmission Control Protocol.

TTP Trusted Third Party.

UDP User Datagram Protocol.

WAL Write-Ahead Logging.

Chapter 1

Introduction

1.1 Reason of research

Blockchain is a network of computers, which each keep their own copy of a chronological, distributed database[1]. A blockchain allows nodes to safely exchange data with other nodes, in an trustworthy, transparent manner. Because of these qualities, there is no need for a Trusted Third Party (TTP) on their integrity, since the blockchain is an entirely self-regulating, decentralized system. A well known project built on blockchain is for instance the Bitcoin, which also happened to be the project which contained the first blockchain[2]. Bitcoin is an example of a public blockchain, where everyone is able to be a participant of the network. In contrast, a private blockchain is a closed blockchain, where the participants are controlled by a certain organization[3]. A company - such as Centric, in this case - could prefer a private blockchain over a public blockchain, if the data they want to put on a blockchain is sensitive to public exposure, but they still want to enjoy the qualities of a blockchain.

According to Job Bakker, one of the characteristics of a blockchain is a consensus algorithm[1]. A consensus algorithm in the context of a blockchain allows *nodes* (participants) to reach *consensus* on what data gets added to the distributed ledger of the blockchain. There are several ways of reaching consensus, such as proof-of-work and proof-of-stake[4]. The problem with these ways of validation, is that these are external ways of reaching consensus, meaning that the nodes of the network itself aren't able to reach consensus themselves. Public blockchains usually make use of this type of external validation[2]. On a private network, there is no need for external validation, because all of the participants operate in a controlled, enclosed network. This means that a different way of consensus is required. A potentially useful way of reaching consensus on a private blockchain is with the use of the Paxos algorithm, developed by Leslie Lamport, and appeared in the article "The Part-Time Parliament" [5]. In this article, Leslie Lamport describes a historical, Greek parliament, which had to pass decrees even though not every parliamentary priest could be present at all times. The priests made use of a sturdy ledger (database), wrote decrees with indelible ink (immutability), and no two priests' decrees could contain contradictory information (consistency)[6]. Based on this description, the Paxos algorithm might be the solution to providing a private blockchain with a suitable consensus algorithm.

1.2 Research value & stakeholders

1.2.1 Research value

The Paxos algorithm is a unique algorithm, because there are only few proven real world use cases which use the Paxos algorithm. Aside from that, it is not know if a blockchain has ever implemented Paxos. The goal is to see if it's possible to make a blockchain simulation based on the Paxos algorithm. If a proof of concept has been successfully made, we will be able to see what the Paxos algorithm does, and what else the algorithm can be good for. Having a

working Paxos blockchain simulation can possibly be groundbreaking for the development of blockchain technologies and distributed systems.

Not only can it be beneficial for the previously named fields in computer science, it can also provide Centric with a helpful product; Centric is currently working with multiple companies to implement a working private blockchain, so that these companies are able to share their data without having to rely on a Trusted Third Party. This specific use case is the CO₂ emission of goods in between departure and delivery. If this data is put on a blockchain, there will be no need for a Trusted Third Party to verify the sustainability of the means of transportation, since a blockchain's data is immutable by design. In this case, a private blockchain is needed, since multiple parties will be working together, but they don't want anyone to be able to participate to the network without invitation. Therefore, an internal consensus algorithm is required, which Paxos might be able to fulfill this role

1.2.2 Stakeholders

As mentioned earlier, multiple companies will be working together to discuss the value of a private blockchain containing CO₂ emission data of delivery vehicles. I will briefly mention every stakeholder, and their role in this project.

- **Rotterdam University of Applied Sciences:** Project management.
- **Centric:** Multinational company, providing several Information Technology related services. Centric is responsible for the research and development of the blockchain technologies.
- **DO-IT (Dutch organic international trade):** Importer and exporter of 100% organic food ingredients and consumer goods[7]. DO-IT wants to contribute to a sustainable society, and thus would like to implement such a blockchain for their transportation services.
- **Evofenedex:** Logistics association/company. Primarily concerned with optimizing logistics for members (companies) of the association[8]. Evofenedex will organize several events and report information and knowledge on this project or these technologies in general, and how it can benefit logistics.
- **Van Reenen:** A transportation company, concerned with sustainability[9].

1.3 Company, workplace & tasks

1.3.1 Company

Centric is the company where I will be conducting the research on this topic. Centric is a Dutch, multinational company which provides several services regarding computer software: software solutions, it-outsourcing, business process outsourcing and staffing services[10]. Key sectors include: Dutch government and water boards, education, logistics (e.g. Centric developed a system for more transparency in the supply chain), construction, health care, and finances[11, 12]. Centric is one of the leading all-round ICT companies of the Netherlands.

1.3.2 Workplace & tasks

I will be doing research on Paxos, Blockchain, distributed systems and computer networking. The research will be conducted at the Centric H.Q. in Gouda. With all of the collected infor-

mation, a blockchain simulation¹ will be made, based on the Paxos algorithm of the Part-Time Parliament.

1.4 Goal

The most important task will be to create a working implementation of the Paxos algorithm in .NET, to see what the Paxos algorithm can actually do, and what it can offer. Once the pure Paxos algorithm has been made, the Paxos algorithm can be expanded further with for example Byzantine Fault Tolerance[1, 13], Disk Paxos[14] and Fast Paxos[15], however due to time constraints, these will only be implemented if there is enough time left to do so. Ultimately, creating Paxos and understanding how it works and what it can offer will be the primary goal of this research.

1.5 Problem statement

1.5.1 Paxos

The Paxos algorithm is not an ordinary algorithm; the original article which described this very algorithm - The Part-Time Parliament[5] - was a very confusing document for many[16], most likely due to the cryptic way the algorithm was described. Besides, not many real-world working paxos implementations have been proven to exist (and those which do exist, are proprietary), so to understand what the algorithm can provide for a private blockchain, we need to understand how the Paxos algorithm works, what it contains, and what it can ultimately provide us. A blockchain simulation using the Paxos protocol is needed to show what value the algorithm has.

Paxos is fault tolerant by nature, which means that the blockchain will be able to keep operating if either hardware or software fails[5, 16]. However, another important characteristic of blockchain is Byzantine Fault Tolerance[1]. The Byzantine Generals problem refers to the situation where a malicious entity exists, and spreads misinformation among many, and therefore causes inconsistency in the distributed ledger [13]. This means that in case the part-time protocol is not Byzantine Fault Tolerant, it won't suffice on its own. Research has to be done if (a) The Part-Time Parliament contains any kind of Byzantine Fault Tolerance, or (b) if Byzantine Fault Tolerance can be implemented in any other way, in order to make the blockchain contain the most vital of the characteristics.

An implementation of the Part-Time Parliament protocol will be made, since we want to see what the original Paxos protocol can do. To create an implementation as close as possible to the original algorithm, we need to stay with The Part-Time Parliament, and translate all of the metaphors to code. There is a chance the metaphor will be interpreted incorrectly, and therefore we might build a system which is not the same as the intended system. However, there exists another article written by Leslie Lamport, called: Paxos Made Simple [16]. Essentially it claims to be the virtually same algorithm described in The Part-Time Parliament, however in much simpler words. However, my company supervisor suspected that Lamport might have made changes to the algorithm due to the harsh critique he had received after publication of the Part-Time Parliament. In an email I asked Leslie Lamport himself for verification, in which he responded:

¹Simulation implies that the final product simply will contain the most important characteristics of blockchain. It doesn't have to be a ready-for-production blockchain.

Paxos Made Simple contains an informal description of the same algorithm that is described more precisely in the Part-Time Algorithm. I don't remember if the two papers use exactly the same names for things.

Leslie Lamport

This essentially means that Paxos Made Simple can be used as well to research the Paxos protocol. However, it will most likely be used to make sense of the cryptic metaphor described in the Part-Time Parliament, since Paxos Made Simple is not as precise. Meaning, that any implemented feature of Paxos Made Simple as opposed to the Part-Time Parliament will have to be documented, to make clear what adaptations have been made to the original algorithm in the final product.

Main question: What value does the Paxos algorithm have for reaching consensus in the context of a blockchain?

Sub questions

- What operations does Paxos take?
- What requirements does Paxos consist of?
- How does Paxos ensure the required qualities of a blockchain, such as immutability of the data and reaching consensus?
- How do the metaphors translate to software?
- Is Paxos Byzantine Fault Tolerant?
- Are there any improvements to be made to Paxos?

1.5.2 Network layer

Paxos will have to be built on a structure which will allow nodes to communicate with each other in a decentralized, peer-to-peer, asynchronous way[5, 16, 17] (An asynchronous event or process is able to operate concurrently to other events and processes). Things such as the transport layer protocol, programming language/framework, network participation and asynchronous communication need to be taken into consideration. Preferably it will support most - or at least a large portion - of all devices, since greater overall support means quicker adaptation and easier setup for users.

Sub questions

- How is a stable peer-to-peer network implemented?
- What protocols should be used?
- How do nodes find each other?
- Are there any remaining holes in the network, and how do I improve these?

Chapter 2

Methods

In order to successfully research this complicated subject we will need to identify how the research will be conducted, what the quality requirements are, how these findings will be validated, and how the findings will be transformed into a working proof of concept.

2.1 Research method

There will be two types of research methods used:

1. **Case study:** We will go in depth on the article *Part-Time Parliament*, written by Leslie Lamport. This article will be the central source of information for development of the system.
2. **Applied research:** The information gathered will be directly applied to create a proof of concept.

2.2 Gathering information

The several subsystems each have different requirements; therefore, different sources, and different ways of gathering information will be required for each subsystem.

- **Paxos:** The Part-Time Parliament and Paxos Made Simple, by Leslie Lamport.
- **Further Paxos improvements:** The Byzantine Generals Problem, and later Paxos articles (such as Disk Paxos, Fast Paxos...) written by Leslie Lamport.
- **Network Layer:** Official .NET documentation, reports, articles, interviews.

2.3 Quality requirements

No quality requirements are known.

2.4 Research validation

The network layer and rest of the application will be validated by conducting interviews with knowledgeable people, however Paxos will be difficult to validate; I can only rely on the two articles (1) The Part-Time Parliament, and (2) Paxos Made Simple. It's an obscure algorithm not many people have implemented themselves, therefore I can't really interview people for verification of the implementation. In order to make sure Paxos is built correctly, a careful, systematic approach will be required: analyzing the text, translating the metaphors to normal terms, breaking everything down into smaller requirements, and implementing the requirements one by one. If any requirements are unclear, I will be able to ask my fellow interns, who've also taken a good look at the Paxos algorithm.

2.5 Sources validation

Most - if not all - sources cited will need to have at least some authority on the subject, and all articles and reports need to refer to sources which do have this authority, if they don't have these by themselves.

Any Paxos article written by Lamport is automatically valid, as long as the pure Paxos is built based on The Part-Time Parliament and Paxos Made Simple [5, 16].

2.6 Project method

Development of the project will be done primarily by myself in an agile manner. This means that the project will be separated into multiple requirements, and these requirements will be implemented in an iterative manner. Agile is preferable, because unexpected findings or changes are likely to occur; agile allows you to be flexible and easily adapt if these things happen.

Chapter 3

Examination of The Part-Time Parliament

3.1 The author

In order to understand how and why the Part-Time Parliament was written, we will have to go back to the origin of the paper. The Part-Time Parliament was written by A.M. Turing Award winner Leslie Lamport: a computer scientist and mathematician, who has committed himself for the most part to distributed computing, and has a large list of publications to back this experience up[18, 19]. Some of his well known works before the appearance of The Part-Time Parliament include:

- *Time, Clocks, and the Ordering of Events in a Distributed system*: Synchronizing the ordering of events across a distributed system, with the use of time-stamps[20].
- *The Byzantine Generals Problem*: Handling malfunctioning or corrupt components giving conflicting information to different parts of a system[13].

These articles were found to be very influential for distributed computing, and are among the most cited articles of computer science[21]. His work on these subjects is reflected in the Part-Time Parliament as well: the Paxos protocol is able to handle asynchronous communication. (An asynchronous event or process is able to operate concurrently to other events and processes). One way of handling asynchronicity in Paxos is with the use of a unique id for messages exchanged, as opposed to using time-stamps to track when a message was sent. Aside from this, the Byzantine generals problem is briefly referenced in numerous chapters, however the author doesn't go into much detail and mentions (explicitly in Paxos Made Simple) that the Paxos protocol assumes a Non-Byzantine model[16].

3.2 Story behind the article

In the late 80's there was a fault-tolerant file system built called *Echo*. The way it handled most of its faults was quite complicated, so Leslie Lamport thought he had enough knowledge to prove that what the *Echo* builders were doing was impossible. Instead he accidentally came up with the consensus algorithm called Paxos[22]. The Paxos protocol was described in the article "The Part-Time Parliament", written in 1990, however published 8 years later in 1998 in the ACM Transactions on Computer Systems. The 8 year difference between being written and publishing can be explained due to the way the article was written; a metaphor was used to explain how the algorithm works. Lamport described an ancient, Greek parliament of the island Paxos which "*had to function even though legislators continually wandered in and out of the parliamentary Chamber*"[5].

While being a humorous way of explaining an extraordinary algorithm for distributed systems, editors and people interested in the article in general were left confused, and couldn't see

past the “Greek parable”. Lamport tried submitting the paper to *TOCS* in 1990, however they wanted the story removed in order for it to be published. Lamport was frustrated because of this, and thus did nothing with the paper. It wasn’t until many years later he came in contact with people at SRC who needed an algorithm for a distributed system they were building, and Paxos could provide just what they needed. He then tried resubmitting the paper by approaching editors of the *TOCS* once again. Both editors agreed to join in on the joke of the article being “and archaeological discovery”, and “having been found behind a filling cabinet”, for comedic reasons, and to prevent Lamport being forced to rewrite the article without the Paxos parliament metaphor[22]. People were still left confused, and so Lamport wrote *Paxos Made Simple* in 2001, which is an informal, less precise description of the same algorithm[16].

3.3 The Paxos parliament

In *The Part-Time Parliament*, Lamport describes the Greek parliament of Paxos. This parliament contained priests, who each did not want to dedicate themselves to the parliament for an extended period of time. This means that the parliament had to function despite priests continuously coming in and leaving the parliamentary chamber. The current law was determined by the series of decrees which were passed by the parliament. These decrees were written in the priest’s ledger.

3.3.1 Requirements

To ensure that the parliament functioned, certain requirements had to be set.

- No ledger could contain contradictory information.
- In order to ensure *consistency*, decrees were written with *indelible ink*, so no written decree could be changed.
- To guarantee *progress*, a **majority** of priests is required to stay in the chamber for a long enough time until a decree had been passed, and the passed decree would be written in the ledger of every priest inside the chamber.

The previously named requirements could only be met if the priests were given the right tools for the job. Every priest was given the tools for the job (see Table 3.1).

Tool	→	Purpose
A sturdy ledger	→	Recording the current state of the law
A pen	→	Writing decrees in the ledger
Indelible ink	→	Writing unalterable decrees
An hourglass	→	Keeping track of time
Pieces of paper (notes)	→	Reminders of other parliamentary tasks

Table 3.1: The tools each priest used inside the parliamentary chamber, and their purpose

The ledger would contain the current state of the law – that is, every passed decree until now. Written decrees could not be changed, since these were written with indelible ink in a sturdy ledger. Notes were used by the priests to keep track of the progress of the voting in the back of the ledger, and contrary to decrees, notes *are* able to be crossed out. Less important notes were kept on pieces of paper, which could be lost upon leaving the parliamentary chamber. Lastly, priests made use of hourglasses to make sure the voting process didn’t take an indefinite amount of time.

According to the archaeological findings, the chamber of the parliament had poor acoustics, so communication between priests had to be done with the use of messengers, who were able to deliver messages among the priests. A massive problem in the parliament was the unreliability of both priests and messengers; Priests could go on a holiday, forget what they voted on, forget to send a messenger to another priest, etc. Likewise, messengers could forget to deliver a message, deliver a message multiple times, or deliver a message too late. However, when inside the chamber they would always do their work properly. According to Lamport “*an atmosphere of mutual trust prevailed*”[5], which means that while inside the chamber, all of the priests and messengers trusted each other fully. Nonetheless, corrupt, and clumsy (but honest) priests were part of the parliament. We go into more detail on this subject in Section 3.5.5.

Because the parliament couldn’t rely on the presence of all priests to vote, the parliament required a *majority* of all priests to be present in the parliamentary chamber¹. A *majority* of all priests ensures that there is always one priest present who has all of the latest information written in his decree.

3.3.2 Comparison with distributed databases

Before we continue with the parliament, we will take a look at the entities and requirements named in the Paxon parliament. If you pay attention, you can notice a similarity between the parliament and real world distributed systems. Drawing this comparison is important, because this will give an image of how the protocol could function in a realistic example. Table 3.3 shows the comparison given by Lamport in the Part-Time Parliament, and Table 3.2 is my own interpretation of entities named in the Part-Time Parliament.

Parliament	\longleftrightarrow	Distributed system
Chamber	\longleftrightarrow	Network
Messenger	\longleftrightarrow	Network connection
Message	\longleftrightarrow	Network package
Sturdy ledger	\longleftrightarrow	Database
Indelible ink	\longleftrightarrow	Immutability of database data
Hourglass	\longleftrightarrow	Computer time/time-stamps
Notes (back of ledger)	\longleftrightarrow	Stable storage
Notes (pieces of paper)	\longleftrightarrow	Temporary, unstable storage
Absent priest	\longleftrightarrow	Offline node
Absent messenger	\longleftrightarrow	Package loss

Table 3.2: My interpretation of the comparison between the Parliament of Paxos and distributed systems.

Parliament	\longleftrightarrow	Distributed database
Legislator/priest	\longleftrightarrow	Server/Node
Citizen	\longleftrightarrow	Client program
Current law	\longleftrightarrow	Database state

Table 3.3: Lamport’s description of the similarities between the Parliament and distributed systems, mentioned in section 4.1 of The Part-Time Parliament.

¹There is only a majority if more than half of all known priests are inside the chamber.

3.4 Choosing a decree

For a decree to appear in the ledgers in the first place, the priests had to reach *consensus*, meaning that every priest has to agree on a final single value. Reaching consensus was done with a number of various ballots (a referendum), where priests were able to vote for or not vote² on a single proposed decree. A group of priests was selected to vote in the ballot (the quorum), and the ballot would succeed only if every priest of the quorum has voted for the decree. Remember that a quorum exists of a majority of all priests, which means that a ballot could only take place if a majority of all priests is present in the chamber.

A ballot consists of:

- A decree (the proposed value being voted on).
- The ballot's quorum.
- The quorum's priests whom **voted for** the decree, after the voting has taken place.
- A unique ballot id.

A unique ballot id is required, since multiple ballots could take place simultaneously due to the asynchronous nature of the parliament. The parliament required older ballots (that is, ballots with a lower id) to be cancelled, since it doesn't represent the current state of the voting process.

A ballot has three constraints:

1. Every ballot in the series of ballots has a unique id.
2. The quorums of every ballot has at least one priest in common (this is done with requiring a majority of priests to be present when a ballot occurs).
3. If any priest voted for a decree in an earlier ballot, the value of the current decree will equal the latest of those previously voted decrees.

The third requirement makes sure that once a value has been voted for by all quorum's priests in an earlier ballot, but failed to be written on the decrees (due to forgetfulness, vacant priests, missing messengers, whatever else can go wrong), that this information has not been lost. Priests made sure to scribble their vote down in the back of the ledger to make sure they won't forget what they last voted for.

²There is **no** voting against a certain decree in a ballot.

#	decree	quorum and voters			
2	α	A	B	Γ	Δ
5	β	A	B	Γ	E
14	α		B		Δ E
27	β	A		Γ	Δ
29	β		B	Γ	Δ

Figure 3.1: Example of a series of ballots. Ballot 27 is the successful ballot, which will cause the decree β to be written to the ledgers of all priests in the chamber. This is because every priest of the quorum voted for the decree. The successful ballot’s chosen decree is β , since that is the last decree a priest of ballot 27’s quorum voted for. In this case, Priest Δ voted for α in ballot 2, but since priest Γ voted for decree β in ballot 5 (a later ballot), the value of ballot 5 is chosen.

This method of reaching consensus with use of balloting is based on the complete version of the single-decree Paxos protocol, the *Synod Protocol*, which I will describe in more detail in section 3.5.3.

3.5 Versions of the protocol

The Paxos protocol went through a series of different versions until the final Parliamentary Protocol was invented. First, a set of *three* constraints was made which would guarantee consistency in the voting process, however would not guarantee progress (these were the three requirements of the balloting process mentioned earlier). Based on these requirements, the *preliminary protocol* was made. To make things more simple for the busy priests, a more restricted version of the preliminary protocol was made, the *basic protocol*, which keeps the *consistency* trait of the preliminary protocol, however allows the priests to keep track of less and achieve the same result. From the basic protocol, the complete *synod protocol* was made, which would guarantee both *consistency* and *progress*. Lastly, ordinary parliaments require a subsequent series of decrees to be passed. All protocols up until the synod protocol were made to come to consensus for just *one* single decree. Therefore, the final Paxos protocol, called the *Multi-decree protocol*, allows for a series of decrees to be written in the distributed ledger.

Preliminary	→	Basic	→	Synod	→	Multi-decree
Consistency	→	Consistency	→	Consistency Progress	→	Consistency Progress Multiple decrees

3.5.1 Preliminary protocol

The preliminary protocol was derived from the set of constraints mentioned in 3.4. Under these constraints, the preliminary protocol guarantees consistency during the balloting process.

The first constraint required every ballot to have their own unique id. A priest could remember what ballot id's he had used for previous ballots by keeping track of this number with notes on the back of his ledger. However this didn't keep different priests from initiating ballots with equal numbers. To prevent different priests from initiating the same ballot id's, ballot id's would consist of two parts: (1) a number unique to the priest, and (2) a unique identifier of the priest himself. In the Part-Time Parliament, Lamport uses the name of a priest, however for simplicity, we will use a number. Imagine priest 7 wants to initiate a new ballot, and has the ballot number 67 written on his notes. He can do this by creating a new ballot with the id of *68.7* (68 being an increment of 1 from 67, and 7 being its own priest id). This way, every priest is guaranteed to always create a unique id, because no other priest is able to replicate the priest id of the ballot id.

The second constraint required a ballot's quorum to consist of a majority of all priests. Lamport describes a few methods: (A) a simple majority, (B) presence rating for each priest, and getting a majority of these ratings, and (C) attendance rating for each priest, and getting a majority of this attendance rating. Whatever the method, every ballot's quorum should at least have one priest in common with any other ballot's quorum.

The third and last constraint of the preliminary protocol decides what decree (proposed value) will ultimately be written to the ledger. Since that multiple instances of the Paxos protocol can be executed concurrently, we want to keep consistency of the chosen values. We don't want another Paxos instance to simply overwrite the previously chosen value of a concurrent/earlier instance of the Paxos protocol. Once a priest has cast a vote on a ballot, it will keep a "promise" to not vote for any other decree. So, if any priest of the newer ballot's quorum voted for a decree in any previous ballot, the decree for this current ballot will then be the latest decree one of the priests of the current quorum voted in any previous ballot. Otherwise if there are no such votes, the decree for the current ballot can be any decree: the initially proposed decree. An example of this requirement is given in Figure 3.1. Notice how the successful ballot's decree is taken from a previous ballot in which a quorum member voted.

The protocol's process is divided into a series of 6 steps. Steps 1-4 describe starting a ballot and voting, and steps 5-6 describe how a passed decree will be written to the priests' ledger

1. A ballot is initiated by a single priest p , who creates a new ballot number b (in accordance with requirement 1 of the Preliminary Protocol) and sends a *NextBallot(b)* message to a number of priests Q (depends on implementation, but on a smaller scale you could take every online node).
2. Priest q – belonging to the set of priests – receives a *NextBallot(b)* message, and replies with a *LastVote(b, v)* message, where v is priest q 's last vote q voted for less than b^3 . If q did not vote for any previous ballots, priest q will return a *null vote*.
3. If priest p had received all *LastVote(b, v)* messages from every priest in Q , priest p will then initiate a new ballot with ballot number b , quorum Q and the decree d , which is chosen based on requirement 3 of the preliminary protocol. Priest p then scribbles a note in the back of the ledger as a reminder that he last tried a ballot with a number b . Afterwards, he begins collecting votes by sending a *BeginBallot(b, d)* to every priest in quorum Q .

³A *LastVote(b, v)* message is a promise to not vote in any ballots with id's between v 's *id* and b .

4. When priest q receives a *BeginBallot*(b, d) message, he will decide whether or not to cast his vote for the current ballot. A priest could choose not to vote for a ballot, if, for instance, the received ballot number b is lower than the ballot number priest q promised to vote for. If priest q decides to vote for the decree, he will send a *Voted*(b, q) message to priest p and writes the vote in the back of his ledger.
5. If priest p receives a *Voted*(b, q) from all priests of quorum Q , he will send a *Success*(d) message to every present priest, including himself.
6. Upon receiving a *Success*(d) message, priest q will record the successful decree in his sturdy ledger with indelible ink.

With the use of the requirements in the steps of the preliminary protocol, consistency is remained in the balloting process, since once a value has been chosen, no other value can overwrite this value.

3.5.2 Basic protocol

The basic protocol is a restricted version of the preliminary protocol, and allowed the priests to keep track of less information, yet achieve the same result as the preliminary protocol. Therefore, the *consistency* requirement is not lost. In the preliminary protocol, a priest p had to keep track of: (1) all ballot ids of which he tried to start, (2) all votes he had cast, and (3) all *LastVote* messages he has sent (which is an agreement to participate in a ballot). In the basic protocol, p had to only record these in the back of his ledger:

Variable	Information
<i>lastTried</i>	The ballot id of the last ballot p attempted to initiate, or $-\infty$ if he never did.
<i>prevVote</i>	The vote of the last ballot p voted for, or $-\infty$ if he never did.
<i>nextBal</i>	The ballot id of the last ballot p promised to participate in, or $-\infty$ if he never did.

In the preliminary protocol, priest p is allowed to hold concurrent ballots. This is not allowed in the basic protocol, which allows only one ballot at the time to be held by p : the ballot with ballot id *lastTried*. Ballot id's lower than *lastTried* are ignored by p in the basic protocol, and a *LastVote*(b, v) message is now a more strict promise to not vote in any ballots with id's lower than b .

The steps taken by the basic protocol are similar to that of the preliminary protocol, however some changes have been made. I cite Lamport, since this requires a detailed explanation, of which the original explanation is the most clear and concise[5]:

1. Priest p chooses a new ballot number b greater than *lastTried*[p], sets *lastTried*[p] to b , and sends a *NextBallot*(b) message to some set of priests.
2. Upon receipt of a *NextBallot*(b) message from p with $b > \textit{nextBal}$ [q], priest q sets *nextBal*[q] to b and sends a *LastVote*(b, v) message to p , where v equals *prevVote*[q]. (A *NextBallot*(b) message is ignored if $b \leq \textit{nextBal}$ [q].)
3. After receiving a *LastVote*(b, v) message from every priest in some majority set Q , where $b = \textit{lastTried}$ [p], priest p initiates a new ballot with number b , quorum Q , and decree d , where d is chosen to satisfy requirement 3 of the preliminary protocol. He then sends a *BeginBallot*(b, d) message to every priest in Q .

4. Upon receipt of a *BeginBallot*(b, d) message with $b = \text{nextBal}[q]$, priest q casts his vote in ballot number b , sets $\text{prevVote}[q]$ to this vote, and sends a *Voted*(b, q) message to p . (A *BeginBallot*(b, d) message is ignored if $b = \text{nextBal}[q]$.)
5. If p has received a *Voted*(b, q) message from every priest q in Q (the quorum for ballot number b), where $b = \text{lastTried}[p]$, then he writes d (the decree of that ballot) in his ledger and sends a *Success*(d) message to every priest.
6. Upon receiving a *Success*(d) message, a priest enters decree d in his ledger.

3.5.3 Synod Protocol

The preliminary and basic protocol guarantee *consistency*, however they do not guarantee *progress*. To achieve *progress*, some extra requirements had to be made to the basic protocol. To help progress, steps 2-6 have to be executed as soon as possible, however you can't improve progress if there is no progress to be made; that is, if there is no requirement deciding when a ballot should be attempted (step 1). Therefore, to achieve progress we need rules on whenever ballots should be initiated.

Ballots won't always be able to finish if there are too many ballots happening at the same time. Since the priests are required to respond to only the latest ballot, there is no guarantee any ballots will finish. In step 2 a priest could promise to vote for a ballot b , which prevents the priest from voting in step 4 for any other ballot lower than b . Therefore if ballots are conducted too frequently, we cannot ensure any progress since priests could continuously propose new ballots with incremented ballot ids, canceling out the ballots.

So therefore, to achieve progress we want ballots to be conducted as quick as possible, but not too quick. Firstly, we need to know how long it took the priests to deliver messages and respond to messages. (Remember that the priests used a hourglass to keep track of the time!) If a priest would stay inside the chamber, it would always deliver a message in 4 minutes, and receive a message in 7 minutes. Thus, if a priest p sent a message to priest q , p could expect a reply from q in 22 minutes, assuming both don't leave the chamber.⁴

If just a single priest p was executing a ballot, he should expect every quorum member to respond in 22 minutes for both the *LastVote*(b, v) messages in step 2, and *Voted*(b, q) messages in step 4 in order to continue the ballot. If some priests didn't respond, then either (a) one or more priests left the chamber, or (b) a different ballot was initiated with a higher ballot id. If a quorum priest left the chamber, priest p would be required to initiate a new ballot. Otherwise, if a priest p learned about a higher numbered ballot, he'd begin a new ballot with a ballot number higher than the cancelled ballot's number. One method of priest p to learn about newer ballot id's, is to require priest q to send *nextBal*[q] to priest p , when q received a *NextBallot*(b) or *BeginBallot*(b, d) from p with $b < \text{nextBal}[q]$. Priest p will then begin a new ballot with a larger ballot number than *nextBal*[q].

Assuming just one priest is still conducting ballots, he could expect a ballot to be completed within 99 minutes, since it could take 22 minutes to start a new ballot, 22 more to learn about higher ballot id, and then 55 more minutes to finish step 1-6 (22 for waiting on the *LastVote*(b, v) messages in step 2, 22 minutes waiting for the *Voted*(b, q) messages in step 4, and 11 minutes for sending *Success*(d) messages in step 5). Hence why we need only a single priest conducting ballots at the same time.

With the Synod protocol, the parliament would have a single priest p to be the designated *president*, who would be the only priest able to initiate new ballots. Having a single priest

⁴These time spans have been taken literally from The Part-Time Parliament, and do not represent actual time. In section 4.3.3 I will discuss my findings on translating the Paxos time into actual time for my implementation of the Paxos algorithm.

conducting ballots prevents most cases of conflicting ballot ids, since only a single priest's ballot id ($lastTried[p]$) is being incremented. Although there are possibly multiple ways of picking a leader, a simple way of doing so is alphabetically (or in case of using numbers, the lowest id would be the president). Every priest sends a message to every other priest after every $T - 11$ minutes. If a priest hasn't received a lower priest's identifier in T minutes – say, priest B didn't get a message from priests C and D in T minutes – then priest B could consider himself to be president, and is now able to conduct ballots. These messages also contain the $lastTried$, so if priest $lastTried[p] > lastTried[q]$, priest q would then write $lastTried[p]$ in the back of his ledger, overwriting $lastTried[q]$. That way every priest in the chamber would always have the latest ballot id written in the back of the ledger as $lastTried$.

Altogether, the *Synod* protocol is a protocol guaranteeing *consistency* since it was derived from the basic protocol, and *progress* by keeping track of time while conducting ballots, and having a single designated *president*.

3.5.4 Multi-decree protocol

Up until now, the Parliament was only able to reach consensus over one single decree, and this single decree was written in ledgers which could contain at most one decree. For a parliament which had to govern a lively island, one law obviously wasn't enough. Thus, the multi-decree protocol (officially called *The Multi-Decree Parliament*) was derived from the complete Synod protocol, which would allow the parliament to write multiple decrees in the distributed ledger. Some adaptations had to be made to the Synod protocol, in order for the parliament to come to consensus on a series of subsequent proposed decrees.

Instead of just one president being able to propose new decrees, non-presidents would be able to propose new decrees by redirecting their decree to the president, who would then execute the entire multi-decree protocol. Execution of steps 1 and 2 of the Synod Protocol occurs just once, since steps 1-2 (if run successfully) cause the quorum members to promise to vote for the president's next ballot; the balloting actually begins from step 3, and assuming there is only one president conducting ballots at the same time, no conflicting ballot id's will exist, unless whenever a new president is selected. (Which will then execute steps 1-2 once more to prevent conflicting ballot ids.) From there on, the president is able to skip steps 1-2, and conduct new ballots by executing steps 3-6 every time it receives a new decree to write to the ledger (from either a citizen, himself, or a different priest).

Whenever a priest was assigned to be the president – in this case, priest p – he would take note of n : the last decree id, up until a missing decree,⁵ or the id of the last written decree if no decrees are missing in the ledger. He would then proceed to send just one *BeginBallot*(b, n) message to a set of priests Q . A priest q could then reply with a *LastVote* message, containing (1) the usual *LastVote* information, and (2) all entries (decree id + decree) in q 's ledger with a higher decree id than n , and q would then ask p of any decrees in p 's ledger with decree id's equal to or lower than n , not written in q 's ledger. Once having received all *LastVote* messages from every priest in Q , p could then immediately begin balloting for every decree learned by p with the *LastVote* messages sent by Q 's priests. The choice of decree for these ballots is determined by requirement 3 of the preliminary protocol.

Once the balloting of the learned decrees was completed successfully, a president could still have gaps in his ledger — that is, missing decrees. Failure of the president could cause a gap in the ledger. (E.g. a president could propose a decree before it learned a previous decree has been chosen.) Anyhow, to solve this problem, president p could attempt to fill a gap by attempting

⁵Priest p – like any other priest – was prone to forgetfulness, would often take naps during important ballots, and spent more time on holiday, rather than taking responsibility for his parliamentary tasks.

to pass an unimportant decree, which would make no difference for any other parliamentary priest:

The ides of February is national olive day

After the gaps in the ledger have been filled with *olive day decrees*, president p would then be ready receive any newly proposed decree from a citizen or a different priest, assign a new decree id to said decree, and attempt to start a new ballot. He would lead the parliament until the inevitable unavailability of the priest, and a new president had to be assigned. From there, the newly assigned president starts out by sending a *BeginBallot* message, and the entire process of the *Multi-decree protocol* is repeated once again.

3.5.5 Byzantine Failures

In the Part-Time Parliament, Lamport describes multiple ways to improve the multi-decree protocol, and mentions the known shortcomings of the Paxos algorithm. Among these is dealing with Byzantine Failures.[5, 13] A Byzantine Failure is the issue where at least one node of a network application is faulty - either intentionally or unintentionally - and sends corrupt/imperfect/malicious information to other nodes. To describe this problem, Lamport used another metaphor in his article *The Byzantine Generals Problem*: a group of generals of the Byzantine army have to agree upon a battle plan, however one or more of the generals might be a traitor, trying to confuse the army by giving conflicting messages.[1, 13] According to Lamport, the Byzantine Generals Problem is solvable if at least two-thirds of the generals are honest and loyal, communicating with unforgeable written messages to determine if conflicting information is sent, and to see what the source of that message is. That way, malicious nodes can be identified in a network of nodes.

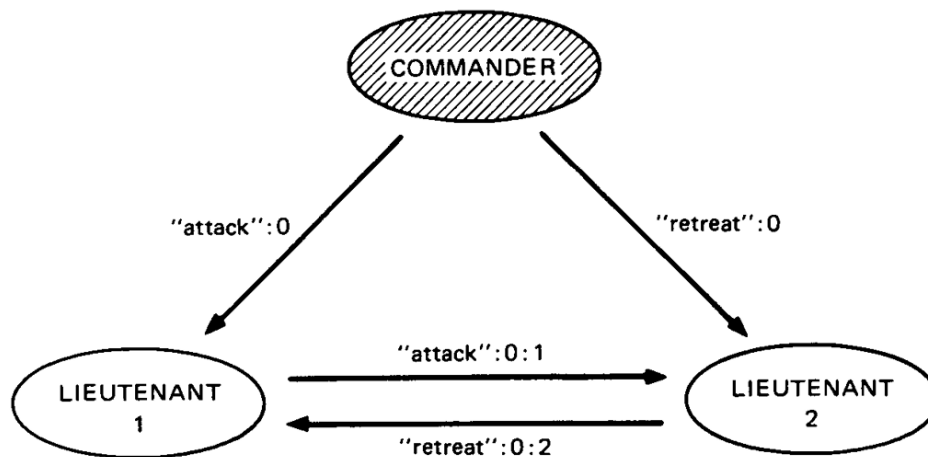


Figure 3.2: The commander (traitor) sent conflicting commands to both lieutenants. By exchanging messages among lieutenants, they can figure out if the received command was honest.

It appears that the Paxos Parliament described in the Part-Time Parliament did not know how to deal with Byzantine Failures, even though it is considered a critical component of the futuristic concept of blockchain.[1] Inside the parliamentary chamber, an atmosphere of mutual trust prevailed, however this naivety goes at the expense of the integrity of the exchanged messages and written decrees, and could certainly cause inconsistency. Corrupt priests likely were part of the parliament at some point, and even honest priests could occasionally make mistakes and send conflicting messages among the other priests. The problem of determining

these corrupt priests or honest mistakes, is if no one is aware of these inconsistencies in the first place.

One method described by Lamport to prevent inconsistency under these circumstances is to have all priests periodically go through the current state of the law (assuming a state-machine implementation), and overwrite the current state with a previously known, verified state. This would give the parliament the *self-stabilizing* property. Unfortunately, it is not exactly known (at the time of writing The Part-Time Parliament) how this self-stabilization actually worked.

3.6 Recap

So far, we have taken a look at the history of the Paxos algorithm, how it was developed, deciphered the metaphor used in the article, taken an in-depth look of the functionality of the various versions of the algorithm, and briefly looked at the Byzantine generals problem (and how the Parliament possibly dealt with this problem). In the next chapter, we will observe how the *synod protocol* and the *multi-decree protocol* were built, based on the Part-Time Parliament and Paxos Made Simple.

Chapter 4

Paxos blockchain simulation development

4.1 Introduction

Building the Paxos algorithm is not solely building the Paxos algorithm; it also meant having to build a foundation for the Paxos nodes to operate on: a network layer. Therefore, the implementation of the assignment is split into two parts: (1) building a network layer, and (2) building the Paxos consensus protocol on top of the network layer. My company supervisor recommended me to write the application in .NET for two reasons:

1. Centric has a lot skilled .NET programmers. This allows me to seek for help easily if needed. Besides, if Centric decides to do something with the application I've built, it will be much easier to integrate it with existing systems, since many systems built by Centric are written in .NET.
2. The CoCo framework[23, 24] (a blockchain framework made by Microsoft, containing Paxos) will be released in early 2018, which will probably be made in .NET. If Paxos is written in .NET, it will be much easier to compare Microsoft's implementation of Paxos.

A decision was made to write the application in .NET core, so that the user has freedom of choice which operating system to run the application on; restricting the application to one specific platform is not beneficial for the flexibility of deployment. .NET core runs on Windows, Mac, and Linux-based systems. No official .NET core documentation was found related to Paxos or consensus algorithms, so this is an implementation written in plain .NET core code.

4.2 Developing the network layer

4.2.1 Requirements

Development of the application began with a network layer. This network layer would allow nodes to communicate with each other in a peer-to-peer (P2P) manner, which means that every node communicates with each other. In order to build the network layer, we will have to determine the requirements for this subsystem. These requirements will be the guideline for building this network layer, and are derived from The Part-Time Parliament and Paxos Made Simple, unless mentioned otherwise. (Note the citations.)

Requirement	Reason
Peer-to-peer	<ol style="list-style-type: none"> 1. <i>“Copies of the database are maintained by multiple servers.”</i>[5] 2. <i>“An implementation that uses a single central server fails if that server fails. We therefore instead use a collection of servers, each one independently implementing the state machine.”</i>[16]
Package corruption	<ol style="list-style-type: none"> 1. <i>“A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again.”</i>[5] 2. <i>“Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.”</i>[16]
Asynchronous messaging	<ol style="list-style-type: none"> 1. <i>“Its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment”</i>[5] 2. <i>“Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model”</i>[16]
Performance	<ol style="list-style-type: none"> 1. <i>“Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.”</i>[16]
Duplicated packages	<ol style="list-style-type: none"> 1. <i>“A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again.”</i>[5] 2. <i>“Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.”</i>[16]
Package loss	<ol style="list-style-type: none"> 1. <i>“A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again.”</i>[5] 2. <i>“Thus, the protocol guarantees consistency even if priests leave the chamber or messages are lost.”</i>[5] 3. <i>“Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.”</i>[16]

Table 4.2: The requirements of the network layer, taken from *The Part-Time Parliament*[5] and *Paxos Made Simple*. [16]

Based on these characteristics we can determine the following requirements of the network layer:

- The protocol allows for peer-to-peer communication.
- Corruption is **not acceptable**.
- Occasional package loss **is acceptable**.
- No guarantee of package ordering **is acceptable**.
- Communication should be done asynchronously.

Preferably the network traffic will be as lightweight as possible to avoid scalability issues, should the application be used by a large network of nodes.

4.2.2 Choosing the right transport layer

Transferring packets between two applications between different computers is typically done using a transport layer. A transport layer is the 4th layer of the OSI model, and is responsible for host-to-host communication between applications[25, 26]. Although there are many types of transport layers, the mainly used protocols in machine-to-machine networking are: Transmission Control Protocol (TCP), and User Datagram Protocol (UDP)[26, 27]. A careful examination of these protocols is required to determine what transport layer is the right choice for the application. If TCP *and* UDP both don't meet the requirements, we will examine alternative transport layers.

Transmission Control Protocol (TCP) ensures a reliable, ordered stream of data between two nodes. It is a connection-oriented transport layer, which means that a secure communication session is established before any data is able to be transmitted between the two parties[28, 29]. With TCP, packets arrive in order, and unrarrived packages will be resent to the receiving node with the use of time-outs[29]. TCP uses a *header* in the process of packaging data for transfer over the network. This header has *fields*, which is a set of parameters containing information used in the transferring of this package (e.g. source, destination, checksum, etc.) A TCP header can be 20-40 bytes of size[30]. This means that while TCP has a lot to offer, it can give a lot of overhead, and thus cause less than ideal performance[27], especially in Peer-to-peer (P2P) applications. Corruption of data is not possible with TCP, due to the checksum included with the header [27, 30].

Contrary to TCP, User Datagram Protocol (UDP) does not have to establish a secure connection in order to send data from host to host. Instead, UDP is a connection-less protocol using datagrams, which are packages that are capable of transferring messages without having to rely on a stable connection[31]. This is also known as best-effort delivery[26, 27]. UDP is typically used for applications where rapid real-time communication is required, such as online games and audio/video streaming. In such applications is occasional lag (package loss) acceptable, and the ordering of packages is not an issue[31]. Since UDP isn't connection-oriented, it does not guarantee that datagrams will arrive in order, or will arrive at all, however the UDP header does include a checksum to make sure a package doesn't corrupt[31]. UDP headers are a mere 8 bytes of size, since all it includes is: (1) a source port number, (2) a destination port number, (3) the datagram size, and (4) the checksum[30, 31].

Combining the characteristics of both TCP and UDP, we can create an overview of the requirements to fulfill of the network layer, to determine what transport layer suits best.

Requirement	TCP	UDP
No corruption	✓	✓
Lightweight	✗	✓
Package ordering	✓	✗
Guaranteed delivery	✓	✗

Table 4.3: A comparison of the transport layer characteristics.

While delivery and ordering of packages is not guaranteed with UDP, this is not a problem for the Paxos protocol, since Paxos is made to be *Fault Tolerant*. The parliament had to deal with forgetful priests and messengers, and the protocol was made to handle these clumsy traits. Ordering is not necessarily a problem, because important messages (such as ballot numbers) always had a unique id, which determines if these messages were dated or recent. Although the parliament is able to handle unarrived messages, it's not preferable. Therefore, one way to tackle this issue is to build a mechanism on application level which resends messages after a specified time-out, and will stop resending after a number of retries, *or* if it has received a confirmation that said message has arrived correctly for the receiving node. The ultimate deciding factor was (1) the smaller package headers, and (2) the lower complexity of the connectionless nature of UDP. Therefore, the network layer will be built based on the User Datagram Protocol, with extra reliability built on application layer. The idea of retransmitting unconfirmed messages and confirming arrived messages on application layer is based on Reliable User Datagram Protocol (RUDP)[32].

Application
Reinforced UDP
UDP (Transport)
IP (Internet)
Network Interface

Table 4.4: The different layers of the network layer.

4.2.3 Gathering connection information

I began building the network layer with a way for nodes to find one another. To do this, I made a Comma Separated File (CSV), which contained rows of an id, an IP address, and a port number for each and every node. Every node will keep a copy of this list, so therefore every node will be able to use the information to connect with each other.

```
id,ip,port
1,192.168.178.230,10000
2,192.168.178.19,10000
3,192.168.178.12,10000
4,192.168.178.18,10000
```

Figure 4.1: The file containing connection information for every node.

The node is able to see what node they are in the list by retrieving their own IPv4 address from their active network device, and comparing it with all of the other endpoints in the CSV. Every node which is not their own IPv4 address, is a peer.

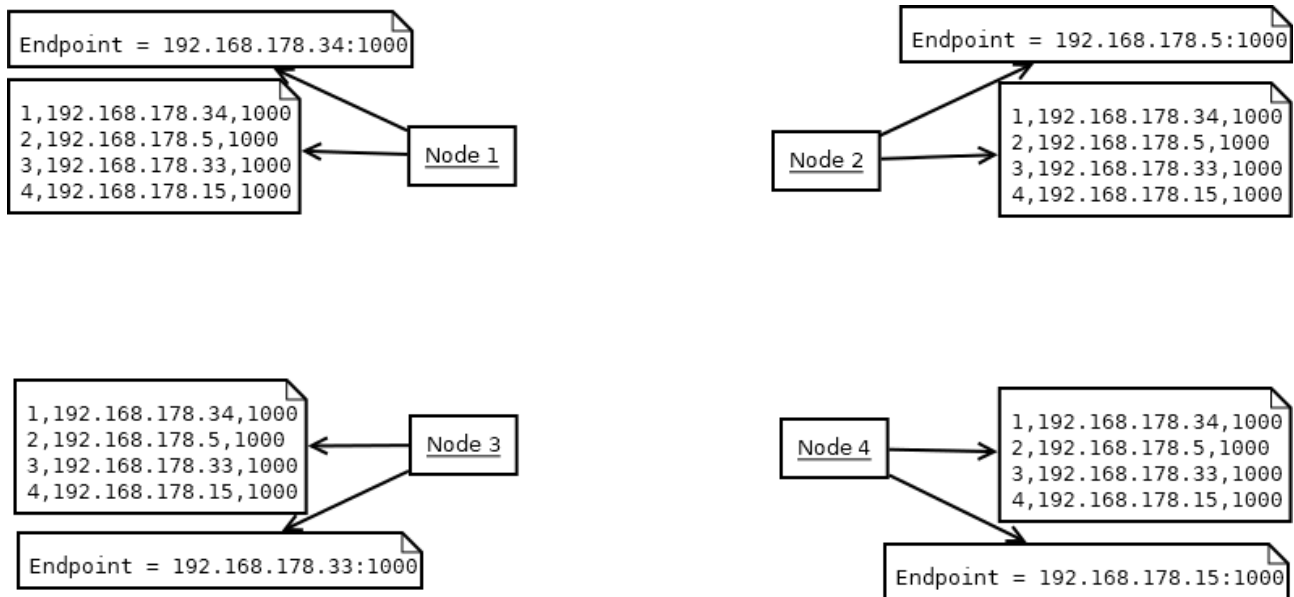


Figure 4.2: An example of nodes with their relevant connection information.

4.2.4 Sending messages between nodes

Subsequently, I decided to split the network communication into two parts: all nodes should fulfill the roles of both a (1) client, and (2) a server, since there is no Trusted Third Party server the nodes can connect with to exchange messages with each other. The client component is responsible for sending messages to peers, and the server component actively listens to messages sent by peers, parses the sent message, and chooses appropriate behavior.

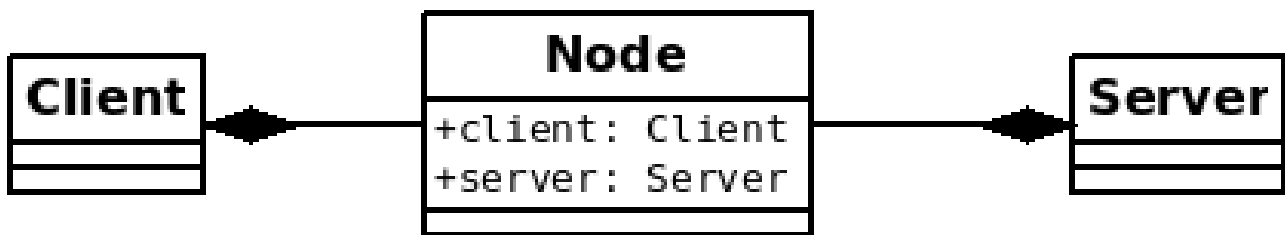


Figure 4.3: Class diagram of a node with a server and client subsystem.

Both the client and server Classes obviously don't do anything without any sort of code, so I wrote some very basic methods to test the network that I was writing: a user could enter a string of text from the command line, which the node would send to all of its peers afterwards. Remember that network communication would be done using the UDP transport layer, so I wrote these methods based around this protocol.

The client continually listens to any input from the user, with use of an asynchronous task. As soon as input has been received, the input will be changed into a *Message Class*. This *Message Class* contains a method to parse this information into a *readable string*, and then to an array of bytes, ready to be sent over the network. Once the message has been prepared, it can be sent from the client to all peers using an *UDPClient*. A *UDPClient* is a .NET Class, which provides utilities for UDP network services, such as asynchronously sending a datagram to another host[33].

At the other end, the server is continually and asynchronously listening to any incoming packages. Once it has received a package, the server has to determine what kind of message this is.¹ The message arrives as an array of bytes, and thus will be converted back into a *readable string*. This readable string contains all of the information of the *Message Class*, such as *id*,² *type*, and content specific to the type of message. (In this case, the content simply contains the string sent from the client of the other node.) Based on the *type* from the readable string, the server can parse this string into a *Message* object, and afterwards determine what behavior we have to choose based on the type of Message it has parsed. In this scenario, it will call a method which will *write* the content of the input from Node *a* to the terminal of Node *b*.

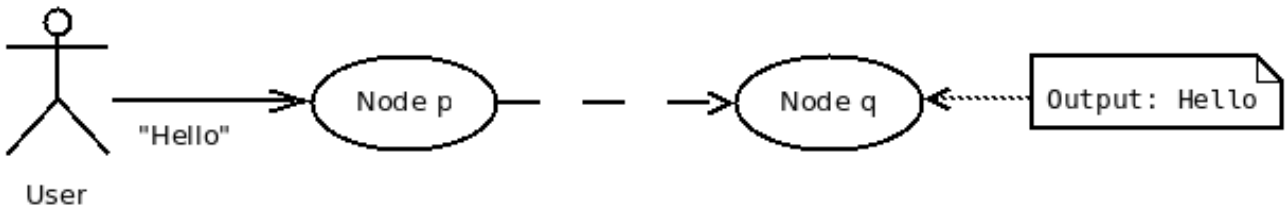


Figure 4.4: A simple example of a message entered by a user at node *p*, sent to node *q*.

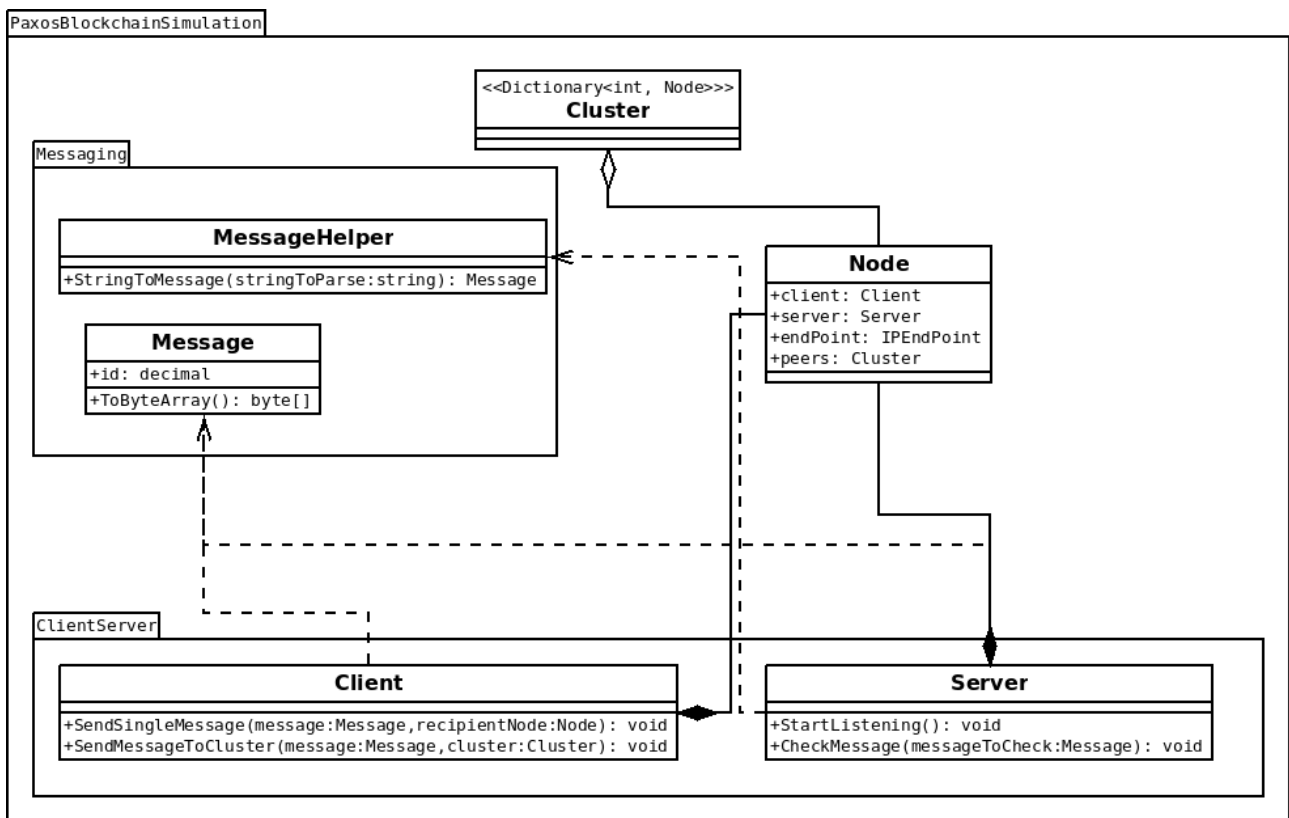


Figure 4.5: A class diagram of the application thus far.

¹As of now it's just a simple string being sent from host to host, however in future development there will surely be many different kinds of messages.

²Message id's are made in the same way ballot id's are generated, such that there are no conflicting message id's from different nodes.

4.2.5 Reinforcing UDP

As mentioned in 4.2.2, one of UDP's differences compared to TCP is that there is no way to determine if packages have been successfully delivered at the receiving host[27, 31]. Occasional package loss is acceptable in our application (see section 4.2.1), however it's still not very preferable. Therefore, we will reinforce UDP at the application level with a mechanism which will confirm packages from the receiving end to the sender, and have another mechanism resend packages from the sender if they haven't been confirmed yet.

The implementation is as follows: Node p keeps track of (1) the messages sent, and (2) the recipients of these messages. After a specified time-out, node p will asynchronously resend the messages to the nodes who didn't respond with a confirmation in time. If node p received a confirmation from node q , node p will then remove node q from the list to resend the messages to. A message will be removed from the to-resend list if (a) every node confirmed the message, or (b) the maximum amount of retransmissions has been reached for this message. In a study on using RUDP in embedded systems, standard UDP gave about 30% package loss, however a package loss of 0% was achieved with a maximum of 3 retransmissions and a time-out of 300ms between retransmission[34]. Hence, these numbers will be used in this implementation as well.³

Occasionally it happens that a message has been resent by node p before node q had the opportunity to respond in time. We don't want our nodes to act on an already received message. Therefore, a temporary log is required to remember what messages the node had already received, so it can decide to do nothing with an already received message, and instead send a confirmation back. This log will be a *queue* of message id's, which will hold a specified amount of message id's it has received in the past.

4.3 Developing Paxos

After the network layer was finished, it became time to build the Paxos protocol on top of it. Thankfully, the Part-Time Parliament contains an Appendix describing the basic protocol in pseudocode[5]. This was a great resource for understanding the workings of Paxos in code. Therefore, I began building Paxos by writing this pseudocode into actual .NET code. The pseudocode describes the functionality of the *basic* protocol. However since the *basic* and *synod* protocol don't differ too much (and to save some space), I will show my .NET implementation of the *synod* protocol, compare that with the pseudocode of the appendix, and mention whatever changes were made between the *basic* and *synod* protocol. Any additional missing functionalities were retrieved from the description of the corresponding versions of the protocol.

4.3.1 Node roles

I began with writing different classes for different roles each priest have, for separation of functionality. Every node has each of these roles, however the state of the nodes decide what roles are active. These roles were derived from *Paxos Made Simple*[1, 16], and their classes will contain the corresponding code of the mentioned functionalities.

- **Proposer:** Proposes new values to write to the distributed ledger. The proposer is able to conduct and manage a ballot, as long as it has the permission to do so (such as being the president).
- **Acceptor:** Is essentially the voter. Decides if it will participate in ballots as a voter and votes for decrees.

³A test case is described in B.1.2, which uses these numbers.

- **Learner:** Learns from proposers whenever a decree has been passed and can be written to the ledger. Also, syncs the ledger's state with the rest of the network's nodes, if its ledger doesn't contain these missing decrees.

4.3.2 Ledgers

4.3.2.1 Manipulating the ledger from code

Since priests had to write on a ledger, a database was required for the priests to write their decrees to. .NET core contains a library called *Entity Framework Core*: an Object-relational mapping (ORM) Framework which functions as a bridge between application and Database. It allows for Create, Read, Update, Delete (CRUD) functions for database manipulation. I chose SQLite as database, since it's easy to implement and deploy. Although if the user prefers, they can easily alter the code to use a different database provider with Entity Framework Core.

EF core makes use of the *DbContext* class to manipulate the data from code. Thus I created a class called *Ledger*, which extends *DbContext* in order for database manipulation. It contains other information such as database source, Tables (Ledger Entries and Progress), preloaded data, etc. Whenever data must be read or manipulated, a new instance of *Ledger* has to be created. From there, data can be read, inserted, and updated. (Assuming it's an unimportant decree.) The *LedgerEntry* Class contains – as the name suggests – ledger entries, written in the database. (All entries are saved in the DbSet *Entries*). Likewise, *PaxosProgress* is a class for saving the important notes in the back of the ledger. The *Progress* DbSet contains only one instance of PaxosProgress.

To prevent database locking – and especially since I chose to use SQLite as our database – Write-Ahead Logging (WAL) has to be enabled on startup of our application. Whenever Write-Ahead Logging is enabled, reading and writing operations can proceed concurrently, without both blocking each other out.[35]

Listing 4.1: The Ledger class and other classes required by the Ledger class.

```

1 namespace PaxosBlockchainSimulation
2 {
3     public class Ledger : DbContext
4     {
5         public DbSet<LedgerEntry> Entries { get; set; }
6         public DbSet<PaxosProgress> Progress { get; set; }
7
8         protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
9         {
10            optionsBuilder.UseSqlite("Data Source=ledger.db");
11        }
12
13        protected override void OnModelCreating(ModelBuilder modelBuilder)
14        {
15            modelBuilder.Entity<PaxosProgress>()
16                .HasData(new PaxosProgress
17                    {
18                        Id = 1,
19                        LastTried = decimal.MinValue,
20                        NextBal = decimal.MinValue,
21                        PrevBal = decimal.MinValue,
22                        PrevDec = new byte[] {}
23                    });
24        }
25    }
26

```

```

27 public class LedgerEntry
28 {
29     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
30     public long Id { get; set; }
31     public string Decree { get; set; }
32
33     public override string ToString()
34     {
35         return string.Format("LedgerEntry: Id= {0}, Decree={1}", Id, Decree);
36     }
37 }
38
39 public class PaxosProgress
40 {
41     public int Id { get; set; }
42     public decimal LastTried { get; set; }
43     public decimal NextBal { get; set; }
44     public decimal PrevBal { get; set; }
45     public byte[] PrevDec { get; set; }
46 }
47 }

```

Listing 4.2: The database Write-Ahead Logging preparation method.

```

1 public class Node
2 {
3     private void PrepareDB()
4     {
5         Console.WriteLine("Preparing DB.");
6
7         using (Ledger ledger = new Ledger())
8         {
9             Console.WriteLine("Connecting with DB.");
10            var connection = ledger.Database.GetDbConnection();
11            connection.Open();
12
13            using (var command = connection.CreateCommand())
14            {
15                Console.WriteLine("Opening DB in WAL.");
16                command.CommandText = "PRAGMA journal_mode=WAL;";
17                command.ExecuteNonQuery();
18            }
19        }
20
21        Console.WriteLine("DB setup successful.");
22    }
23 }

```

4.3.2.2 Immutability

To achieve immutability, some triggers have been set on database (SQL) level. These triggers are set on database migration (that is, when the database is created), and are executed whenever an attempt is made to update or delete an already written ledger. This trigger will return an error code to the user unless an olive-day decree is updated. In theory, this doesn't stop users from inserting new decrees or updating unimportant decrees by hand in the database. I haven't been able to figure out how this can be prevented, however setting user permissions on the database might be a possible solution to this problem. In section B.1.1 these triggers are

tested on functionality.

Listing 4.3: The triggers set for the Entries table.

```

1 public partial class InitDatabase : Migration
2 {
3     protected override void Up(MigrationBuilder migrationBuilder)
4     {
5         //immutability trigger for SQLite. Different DB providers require different SQL
6         //statements
7         migrationBuilder.Sql(String.Format(
8             @"
9             CREATE TRIGGER AttemptUpdate BEFORE UPDATE ON Entries FOR EACH ROW
10            WHEN old.Decree != '{0}'
11            BEGIN
12            SELECT RAISE(ABORT, 'Ledgers are written with indelible ink, and their entries can
13            not be changed');
14            END;
15
16            CREATE TRIGGER AttemptDeletion BEFORE DELETE ON Entries FOR EACH ROW
17            BEGIN
18            SELECT RAISE(ABORT, 'Ledgers are written with indelible ink, and their entries can
19            not be changed');
20            END;
21            ", NodeAgents.Proposer.OLIVE_DAY_DEGREE));
22     }
23 }

```

4.3.3 Implementing the Synod protocol

The Synod Protocol is built by writing the code based on the *Basic* protocol mentioned in the Part-Time Parliament appendix. Afterwards, the other restrictions and features mentioned in the *Synod Protocol* chapter of the Part-Time Parliament were implemented. Any code not described in the appendix was written based on the descriptions of the requirements.

We will now go through the (important) .NET code of the *Synod* protocol – including the 6 steps of an instance of the protocol – and compare it with the pseudocode wherever possible. Extra additions or changes from the pseudocode will be explained whenever necessary.

4.3.3.1 President election

One of the main additions to the Synod protocol is the requirement of having a president, who is the node able to initiate new ballots. For a node to know if it's the current president, the node requires to know the id's of other nodes in the chamber. We will require every node to send a heartbeat (message) to its peers after $T - 11$ Paxos minutes, and make a node a president if it has not received a heartbeat with a node id lower than its own id within T minutes. Remember that T and the Paxos minutes come from the *Synod protocol*, which was missing a description of how much these values actually are. Thus, values had to be determined based on our own experience. (See Table 4.5.)

Action	Paxos time	Real time(ms)
Time between heartbeats	$T - 11$ minutes	$1000 - (11 \times 45.45) = \pm 500$
Waiting for <i>LastVote</i> messages	22 minutes	$22 \times 45.45 = \pm 1000$
Waiting for <i>Voted</i> messages	22 minutes	$22 \times 45.45 = \pm 1000$
President after	T	1000

Table 4.5: Time described as in the Part-Time Parliament, translated to real world, rounded values. Assuming $T = 1000\text{ms}$, and a Paxos minute = 45.45ms, we get values close to 1/2 and 1 second—ideal to work with.

As seen in Table 4.5, whenever a node p hasn't received a message from a node with a lower id within 1000 milliseconds (1 second), it will take the role of president. Any proposals from other nodes will be redirected to the p – which will be explained in section 4.3.3.7.

4.3.3.2 Notes in back of ledger

These are the important things the nodes have to keep track of by writing this information in the back of the ledger. At first these variables were kept solely in Stack and Heap (RAM) and afterwards applied *EF Core* with an *SQLite* database to save the information on disk. A new instance of *Ledger* is created in the *GetLedgerVariables()* method, (as mentioned in section 4.3.2.1), so it retrieves these values from the ledger.

One of the issues I experienced building the Paxos protocol, was not knowing how to implement the decrees. Every use case – and so whatever needs to be written in the decrees – is different, and therefore there is not one single solution on how to build a generic working Paxos protocol. So, to keep things simple, I created one Class called *LedgerEntry*, which contains a string containing the decree. For a *LedgerEntry* to be written in the ledger, the *LedgerEntry* has to be added to the DBSet *Entries*.

Pseudocode

<i>outcome</i> [p]	The decree written in p 's ledger, or BLANK if there is nothing written there yet.
<i>lastTried</i> [p]	The number of the last ballot that p tried to begin, or $-\infty$ if there was none.
<i>prevBal</i> [p]	The number of the last ballot in which p voted, or $-\infty$ if he never voted.
<i>prevDec</i> [p]	The decree for which p last voted, or BLANK if p never voted.
<i>nextBal</i> [p]	The number of the last ballot in which p agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

.NET code

```

1 public class Node
2 {
3     public byte[] _outcome;
4     public decimal _lastTried;
5     public decimal _prevBal;
6     public byte[] _prevDec;
7     public decimal _nextBal;
8

```

```

9   private void GetLedgerVariables()
10  {
11      using (Ledger ledger = new Ledger())
12      {
13          if (ledger.Entries.Count() > 0)
14          {
15              _outcome = MessageHelper
16                  .StringToByteArray(ledger.Entries.Last().Decree);
17          }
18          else
19          {
20              _outcome = null;
21          }
22          _lastTried = ledger.Progress.First().LastTried;
23          _prevBal = ledger.Progress.First().PrevBal;
24          _prevDec = ledger.Progress.First().PrevDec;
25          _nextBal = ledger.Progress.First().NextBal;
26      }
27  }
28  }

```

4.3.3.3 Notes on a slip of paper

These are values the nodes keep track of, though can be lost, which means the variables are only kept in RAM.

Pseudocode

$status[p]$	One of the following values: <i>idle</i> Not conducting or trying to begin a ballot <i>trying</i> Trying to begin ballot number $lastTried[p]$ <i>polling</i> Now conducting ballot number $lastTried[p]$ If p has lost his slip of paper, then $status[p]$ is assumed to equal <i>idle</i> and the values of the following four variables are irrelevant.
$prevVotes[p]$	The set of votes received in <i>LastVote</i> messages for the current ballot (the one with ballot number $lastTried[p]$).
$quorum[p]$	If $status[p] = polling$, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.
$voters[p]$	If $status[p] = polling$, then the set of quorum members from whom p has received <i>Voted</i> messages in the current ballot; otherwise, meaningless.
$decree[p]$	If $status[p] = polling$, then the decree of the current ballot; otherwise, meaningless.

.NET code

```

1   public enum NodeStatus
2   {
3       idle,
4       trying,
5       polling
6   }

```

```

7
8 public NodeStatus _status = NodeStatus.idle;
9 public List<LastVote> _prevVotes;
10 public Cluster _quorum;
11 public Cluster _voters;
12 public byte[] _decree;

```

4.3.3.4 Proposer: Creating ballot id's

These functions fulfill the first requirement of the *preliminary* protocol. It increments the ballot id president p is going to use, and writes it as *LastTried* in the back of the ledger using the *Ledger* class.

.NET code

```

1 public static class MessageHelper
2 {
3     public static decimal CreateUniqueMessageId(long messageId, int nodeId)
4     {
5         return Convert.ToDecimal(String.Format("{0}.{1}", messageId, nodeId),
6             CultureInfo.InvariantCulture);
7     }
8 }
9 public class Proposer
10 {
11     private async Task IncrementBallotId()
12     {
13         if (_parentNode._lastTried == decimal.MinValue)
14         {
15             _parentNode._lastTried = MessageHelper.CreateUniqueMessageId(1, _parentNode._id);
16         }
17         else
18         {
19             _parentNode._lastTried++;
20         }
21
22         await LedgerHelper.SavePaxosProgressAsync(_parentNode);
23     }
24 }

```

4.3.3.5 Proposer: Get Quorum

This function fulfills the second requirement of the *preliminary* protocol. I chose to get all online nodes including self, and do the majority check in the methods using *this* method. The simplicity doesn't impact the functionality of the algorithm, however in a live system you might want to use the more sophisticated methods for obtaining a majority, mentioned in the *preliminary* protocol.

.NET code

```

1 private Cluster GetOnlineNodes()
2 {
3     Cluster quorum = new Cluster(_parentNode.onlinePeers);
4     quorum.Add(_parentNode._id, _parentNode);
5     return quorum;

```

6 | }

4.3.3.6 Proposer: Get ballot decree

This function fulfills the third requirement of the *preliminary* protocol: getting the decree for this ballot. Remember that the decree for the current ballot, is the latest of the already voted for in past ballots (of the nodes in the current ballot's quorum), or can be the proposed decree if there is none.

.NET code

```

1 private byte[] GetDecreeToPropose(byte[] proposedDecree)
2 {
3     LastVote highestLastVote = _parentNode._prevVotes.OrderByDescending(v =>
4         v._prevBal).First();
5
6     if (highestLastVote._prevBal != decimal.MinValue)
7     {
8         return highestLastVote._prevDecree;
9     }
10    else
11    {
12        return proposedDecree;
13    }
14 }

```

4.3.3.7 Proposer: propose a decree

This function continuously waits for user input to propose as a new decree. A round of Paxos is started – with the input being the decree – only if the node is a president, and if there is a majority of nodes online. If the node is not a president, it will redirect its decree proposal to the president, who will then start executing the algorithm, assuming a majority of nodes can still be formed.

.NET code

```

1 public async Task BeginProposingOnInput()
2 {
3
4     Console.WriteLine("[Proposer] Preparing...");
5     while (!_parentNode.client.isSending && !_parentNode.server.isListening)
6     {
7         Thread.Sleep(100);
8     }
9
10    while (_parentNode.client.isSending && _parentNode.server.isListening)
11    {
12        string input = Console.ReadLine();
13        byte[] inputInBytes = MessageHelper.StringToByteArray(input);
14
15        if(_parentNode._isLeader)
16        {
17            ExecutePaxos(inputInBytes);
18        }
19        else if (!_parentNode._isLeader)
20        {

```

```

21     if (_parentNode.leaderNode == null)
22     {
23         Console.WriteLine("[Proposer] Waiting for leader to be known...");
24         while (_parentNode.leaderNode == null)
25         {
26             Thread.Sleep(100);
27         }
28     }
29     DecreeProposal decreeProposal = new
30     DecreeProposal(_parentNode.client._messageIdCounter, _parentNode._id, inputInBytes);
31     await _parentNode.client.SendMessageToNode(decreeProposal, _parentNode.leaderNode,
32         true, true);
33 }
34 }

```

4.3.3.8 Proposer: Step 1 - Execute Paxos

Begins Paxos, and calls the required methods in order of the algorithm. Whenever a new round of Paxos has to be made, this method has to be called. The requirement of atomic execution of the steps allows only one execution of Paxos to occur at once, unless two nodes think they're the president and execute a round of Paxos at the same time.

.NET code

```

1  public async Task ExecutePaxos(byte[] proposedDecree)
2  {
3      if (_parentNode._isLeader
4      && _parentNode._status == NodeStatus.idle
5      && !executingPaxos)
6      {
7          executingPaxos = true;
8          int newBallotMsgResult = 1;
9          int ballotSuccessful = 1;
10         Cluster quorum;
11
12         do
13         {
14             Console.WriteLine("\n[Proposer] Executing Paxos for decree={0}",
15                 MessageHelper.ByteArrayToString(proposedDecree));
16             newBallotMsgResult = 1;
17             ballotSuccessful = 1;
18             newBallotMsgResult = await TryNewBallot();
19
20             if(newBallotMsgResult == 0)
21             {
22                 quorum = GetOnlineNodes();
23                 ballotSuccessful = await StartPollingMajoritySet(quorum, proposedDecree);
24
25                 if (ballotSuccessful == 0)
26                 {
27                     await Succeed();
28                 }
29             }
30         } while ((newBallotMsgResult == 1
31             || ballotSuccessful == 1)
32             && _parentNode._isLeader);
33 }

```



```

34     executingPaxos = false;
35 }
36 }

```

4.3.3.9 Proposer: Step 1 - Trying a new Ballot

Every time a ballot is tried, a new ballot id is created based on requirement 1 of the *preliminary* protocol. This method returns a 1 if the message was unsuccessful, 2 if an error occurred, and 0 if the message arrived in time. In *ExecutePaxos*, *TryNewBallot* will be executed infinitely until success. (So, if *TryNewBallot* returns a 0.)

Pseudocode

Try New Ballot

Always enabled.

- Set *lastTried*[*p*] to any ballot number *b*, greater than its previous value, such that *owner*(*b*) = *p*.
- Set *status*[*p*] to *trying*.
- Set *prevVotes*[*p*] to \emptyset .

.NET code

```

1  public async Task<int> TryNewBallot()
2  {
3      if (_parentNode._isLeader)
4      {
5          await IncrementBallotId();
6          _parentNode._status = NodeStatus.trying;
7          _parentNode._prevVotes = new List<LastVote>();
8          return await SendNextBallotMessage();
9      }
10     return 2;
11 }

```

4.3.3.10 Proposer: Step 1 - Sending a NextBallot Message

Notice how time is being tracked to check the duration between actions. This is one of the requirements for the *progress* condition of the Synod Protocol. (Remember that the Paxon priests were given hourglasses to measure the passage of time.) President *p* has to wait a maximum of 22 Paxon minutes, until he decides that the ballot has failed, and has to be retried.

Pseudocode

Send NextBallot Message

Enabled whenever *status*[*p*] = *trying*.

- Send a *NextBallot*(*lastTried*[*p*]) message to any priest.

.NET code

```

1 public async Task<int> SendNextBallotMessage()
2 {
3     timeAtPreviousAction = DateTime.Now;
4     Cluster setOfNodes = GetOnlineNodes();
5     bool majorityOnline = setOfNodes.HasMajorityOf(_parentNode.allNodes);
6
7     if (_parentNode._status == NodeStatus.trying
8     && _parentNode._isLeader
9     && majorityOnline)
10    {
11        Console.WriteLine("[Proposer] Sending nextBallotMsg to {0}", String.Join(",",
12            setOfNodes.Keys));
13        timeAtPreviousAction = DateTime.Now;
14        NextBallot nextBallot = new NextBallot(_parentNode.client._messageIdCounter,
15            _parentNode._id, _parentNode._lastTried, 0);
16
17        //send message to peers
18        await _parentNode.client.SendMessageToCluster(nextBallot,
19            GetClusterExcludingSelf(setOfNodes), true);
20
21        //send this message to own acceptor
22        await _parentNode.acceptor.OnReceiveNextBallot(nextBallot);
23
24        //wait for reply from set of nodes
25        while (_parentNode._prevVotes.Count() != setOfNodes.Count())
26        {
27            if ((DateTime.Now - timeAtPreviousAction).TotalMilliseconds >=
28                Node.MINUTE_IN_PAXOS_TIME * 22)
29            {
30                Console.WriteLine("[Proposer] Didn't get enough lastvote messages in time.");
31                return 1;
32            }
33        }
34        Console.WriteLine("[Proposer] Received all lastvotes from chosen set of nodes.");
35        return 0;
36    }
37    else if (!_parentNode._isLeader)
38    {
39        Console.WriteLine("[Proposer] Cannot send a NextBallot message as a non-leader.");
40        return 2;
41    }
42    else if (_parentNode._status != NodeStatus.trying)
43    {
44        Console.WriteLine("[Proposer] Cannot send a NextBallot message when not trying.");
45        return 2;
46    }
47    return 1;
48 }

```

4.3.3.11 Acceptor: Step 2 - Receiving a NextBallot Message

This is the method which will be executed when an acceptor receives a *NextBallot* message. If the received ballot id is lower than *nextBal*, then *nextBal* will be sent to the president, who will then retry the ballot with a higher ballot id.

Pseudocode

Receive *NextBallot(b)* Message

If $b \geq nextBal[p]$ then

- Set $nextBal[p]$ to b .

.NET code

```

1 public async Task OnReceiveNextBallot(NextBallot nextBallotMsg)
2 {
3     if (nextBallotMsg._ballotId >= _parentNode._nextBal)
4     {
5         Console.WriteLine("[Acceptor] Received new nextballot message.");
6         _parentNode._nextBal = nextBallotMsg._ballotId;
7         await SendLastVoteMessage(nextBallotMsg);
8         await LedgerHelper.SavePaxosProgressAsync(_parentNode);
9     }
10    else if (nextBallotMsg._ballotId < _parentNode._nextBal)
11    {
12        Console.WriteLine("[Acceptor] Received old nextballot message.");
13        await ReplyNextBal(nextBallotMsg._senderId);
14    }
15 }

```

4.3.3.12 Acceptor: Step 2 - Sending a LastVote Message

Sends the promise to president p that it will participate in this ballot.

Pseudocode**Send *LastVote* Message**

Enabled whenever $nextBal[p] > prevBal[p]$.

- Send a $LastVote(nextBal[p], v)$ message to priest $owner(nextBal[p])$, where $v_{pst} = p$, $v_{bal} = prevBal[p]$, and $v_{dec} = prevDec[p]$.

.NET code

```

1 private async Task SendLastVoteMessage(NextBallot nextBallotMsg)
2 {
3     if (_parentNode._nextBal > _parentNode._prevBal)
4     {
5         LastVote lastVote = new LastVote(_parentNode.client._messageIdCounter,
6         _parentNode._nextBal,
7         _parentNode._id,
8         _parentNode._prevBal,
9         _parentNode._prevDec);
10
11        if (nextBallotMsg._senderId == _parentNode._id)
12        {
13            Console.WriteLine("[Acceptor] Sending LastVote to self");
14            _parentNode.proposer.ReceiveLastVoteMessage(lastVote);
15        }
16        else
17        {
18            Console.WriteLine("[Acceptor] Sending LastVote to president (node {0})",
19                nextBallotMsg._senderId);

```

```

19     Node votingProposer = _parentNode.onlinePeers.GetNodeById(nextBallotMsg._senderId);
20     await _parentNode.client.SendMessageToNode(lastVote, votingProposer, true, true);
21 }
22 }
23 }

```

4.3.3.13 Proposer: Step 2 - Receiving a LastVote Message

If president p receives a *LastVote* message, it will be added to the list of previous *prevVotes* of the current ballot's quorum.

Pseudocode

Receive *LastVote*(b, v) Message

If $b = \text{lastTried}[p]$ and $\text{status}[p] = \text{trying}$, then

- Set $\text{prevVotes}[p]$ to the union of its original value and $\{v\}$.

.NET code

```

1  public void ReceiveLastVoteMessage(LastVote lastVote)
2  {
3      if (lastVote._nextBal == _parentNode._lastTried && _parentNode._status ==
4          NodeStatus.trying)
5      {
6          Console.WriteLine("[Proposer] Received lastvote from node {0}.", lastVote._senderId);
7          _parentNode._prevVotes.Add(lastVote);
8      }
9  }

```

4.3.3.14 Proposer: Step 3 - Start Polling Majority Set

President p received all promises from every node of the current ballot's quorum. It is now time to start balloting. The ballot's values are determined and written on a slip of paper.

The history of ballots (mentioned in Part-Time Parliament as \mathcal{B}) is left out since it is not required for a working protocol, and otherwise it would take significant amount of time to implement for it to sync with other nodes. I concluded that it was not worth it to implement.

Pseudocode

Start Polling Majority Set Q

Enabled when $\text{status}[p] = \text{trying}$ and $Q \subseteq \{v_{pst} : v \in \text{prevVotes}[p]\}$, where Q is a majority set.

- Set $\text{status}[p]$ to *polling*.
- Set $\text{quorum}[p]$ to Q .
- Set $\text{voters}[p]$ to \emptyset .
- Set $\text{decree}[p]$ to a decree d chosen as follows: Let v be the maximum element of $\text{prevVotes}[p]$. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
- Set \mathcal{B} to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = \text{lastTried}[p]$.

.NET code

```

1 private async Task<int> StartPollingMajoritySet(Cluster quorum, byte[] proposedDecree)
2 {
3     bool quorumMembersAreLastVoters =
4     quorum.Keys.ToList()
5     .Intersect(_parentNode._prevVotes.Select(lv => lv._senderId).ToList())
6     .Count() == quorum.Count();
7
8     if (_parentNode._isLeader
9     && _parentNode._status == NodeStatus.trying
10    && quorumMembersAreLastVoters)
11    {
12        Console.WriteLine("[Proposer] Polling...");
13        _parentNode._status = NodeStatus.polling;
14        _parentNode._quorum = quorum;
15        _parentNode._voters = new Cluster();
16        _parentNode._decree = GetDecreeToPropose(proposedDecree);
17        _parentNode._entryId = 1;
18        return await SendBeginBallotMessage();
19    }
20    return 1;
21 }

```

4.3.3.15 Proposer: Step 3 - Sending a BeginBallot Message

Here once more president p has to wait 1 real-time second for the *Voted* messages to arrive. If he hasn't received all *Voted* messages, the ballot is then retried with a higher ballot id (see *ExecutePaxos* method).

Pseudocode**Send *BeginBallot* Message**

Enabled when $status[p] = polling$.

- Send a *BeginBallot*($lastTried[p]$, $decree[p]$) message to any priest in $quorum[p]$.

.NET code

```

1 private async Task<int> SendBeginBallotMessage()
2 {
3     if (_parentNode._status == NodeStatus.polling && _parentNode._isLeader)
4     {
5         timeAtPreviousAction = DateTime.Now;
6         BeginBallot beginBallotMsg = new BeginBallot(_parentNode.client._messageIdCounter,
7             _parentNode._id, _parentNode._lastTried, _parentNode._decree);
8         await _parentNode.client.SendMessageToCluster(beginBallotMsg,
9             GetClusterExcludingSelf(_parentNode._quorum), true);
10
11         //send this message to own acceptor
12         await _parentNode.acceptor.OnReceiveBeginBallot(beginBallotMsg);
13
14         while (_parentNode._voters.Count() < _parentNode._quorum.Count())
15         {
16             //wait for every quorum member to reply
17             if ((DateTime.Now - timeAtPreviousAction).TotalMilliseconds >=
18                 Node.MINUTE_IN_PAXOS_TIME * 22)

```

```

16     {
17         return 1;
18     }
19 }
20 return 0;
21 }
22 else
23 {
24     Console.WriteLine("[Proposer] Cannot send beginballot message, because not polling.");
25     return 2;
26 }
27 return 1;
28 }

```

4.3.3.16 Acceptor: Step 4 - Receiving a BeginBallot Message

A quorum member q can here decide to cast its vote, or inform the president with its written $nextBal[q]$. Remember how in the Synod protocol, if the *BeginBallot*'s ballot id is lower than q 's $nextBal$, then instead of a *Voted* message, q will send its $nextBal$ to p .

Same as in step 3; the history value \mathcal{B} is left out, since requiring other nodes to sync up the values takes additional work, and is not very beneficial to the implementation.

Pseudocode

Receive *BeginBallot*(b, d) Message

If $b = nextBal[p] > prevBal[p]$ then

- Set $prevBal[p]$ to b .
- Set $prevDec[p]$ to d .
- If there is a ballot B in \mathcal{B} with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of \mathcal{B} be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

.NET code

```

1  public async Task OnReceiveBeginBallot(BeginBallot beginBallotMsg)
2  {
3      Console.WriteLine("[Acceptor] Received beginballot [{0}] from {1}",
4          beginBallotMsg._ballotId,
5          beginBallotMsg._senderId);
6
7      if (beginBallotMsg._ballotId == _parentNode._nextBal && _parentNode._nextBal >
8          _parentNode._prevBal)
9      {
10         Console.WriteLine("[Acceptor] Voting for ballot...");
11         _parentNode._prevBal = beginBallotMsg._ballotId;
12         _parentNode._prevDec = beginBallotMsg._decree;
13         await SendVotedMessage(beginBallotMsg);
14         await LedgerHelper.SavePaxosProgressAsync(_parentNode);
15     }
16     else if (beginBallotMsg._ballotId < _parentNode._nextBal)
17     {
18         await ReplyNextBal(beginBallotMsg._senderId);
19     }

```

```

20
21 private async Task ReplyNextBal(int peerId)
22 {
23     Console.WriteLine("[Acceptor] Sending nextBal to president {0}.", peerId);
24     UpdateBallotNumber newBallot = new
25         UpdateBallotNumber(_parentNode.client._messageIdCounter, _parentNode._id,
26             _parentNode._nextBal);
27     await _parentNode.client.SendMessageToNode(newBallot, peerId, true, true);
28 }

```

4.3.3.17 Acceptor: Step 4 - Sending a Voted Message

A quorum node q – or p if he himself is the president – sends his vote to the president.

Pseudocode

Send Voted Message

Enabled whenever $prevBal[p] \neq -\infty$.

- Send a *Voted*($prevBal[p]$, p) message to $owner(prevBal[p])$.

.NET code

```

1 private async Task SendVotedMessage(BeginBallot beginBallotMsg)
2 {
3     if (_parentNode._prevBal != decimal.MinValue)
4     {
5         Voted voted = new Voted(_parentNode.client._messageIdCounter,
6             _parentNode._id,
7             _parentNode._prevBal);
8
9         if (beginBallotMsg._senderId == _parentNode._id)
10        {
11            //send to self
12            Console.WriteLine("[Acceptor] Sending voted to self");
13            _parentNode.proposer.ReceiveVotedMessage(voted);
14        }
15        else
16        {
17            //send to president
18            Console.WriteLine("[Acceptor] Sending voted to {0}", beginBallotMsg._senderId);
19            Node president = _parentNode.peers.GetNodeById(beginBallotMsg._senderId);
20            await _parentNode.client.SendMessageToNode(voted, president, true, true);
21        }
22    }
23    else
24    {
25        Console.WriteLine("[Acceptor] Cannot cast vote because _prevBal has not been set.");
26    }
27 }

```

4.3.3.18 Proposer: Step 5 - Receiving a Voted Message

The ballot's president receives a vote from a quorum node, and takes note of this vote by recording the node's vote in $voters[p]$.

Pseudocode**Receive $Voted(b, q)$ Message**

If $b = lastTried[p]$ and $status[p] = polling$, then

- Set $voters[p]$ to the union of its old value and $\{q\}$

.NET code

```

1 public void ReceiveVotedMessage(Voted voted)
2 {
3     if (voted._ballotId == _parentNode._lastTried && _parentNode._status ==
4         NodeStatus.polling)
5     {
6         Console.WriteLine("[Proposer] Received voted message from {0} for {1}.",
7             voted._senderId, voted._ballotId);
8         _parentNode._voters.Add(voted._senderId,
9             _parentNode.allNodes.GetNodeById(voted._senderId));
10    }
11 }

```

4.3.3.19 Proposer: Step 5 - Success

A ballot has been successfully completed. President p writes this information in his ledger. Afterwards, every Learner of every node of the quorum will be notified with this new decree.

At first I combined both *Success* and *SendSuccessMessage* into one method for simplicity, however in order to stay as close as possible to the original description I decided to separate these two into the intended methods.

Pseudocode**Succeed**

Enabled whenever $status[p] = polling$, $quorum[p] \subseteq voters[p]$, and $outcome[p] = BLANK$.

- Set $outcome[p]$ to $decree[p]$.

.NET code

```

1 private async Task Succeed()
2 {
3     timeAtPreviousAction = DateTime.Now;
4     bool quorumMembersAreVoters =
5         _parentNode._quorum.Keys.ToList()
6         .Intersect(_parentNode._voters.Keys.ToList())
7         .Count() == _parentNode._quorum.Count();
8
9     byte[] outcome = await LedgerHelper.GetOutcome();
10
11     if (_parentNode._status == NodeStatus.polling
12         && quorumMembersAreVoters
13         && outcome == null)
14     {
15         Console.WriteLine("[Proposer] Ballot [{0}:{1}] succeeded.",
16             _parentNode._entryId,

```



```

17     MessageHelper.ByteArrayToString(_parentNode._decree));
18
19     Success success = new Success(_parentNode.client._messageIdCounter,
20     _parentNode._id,
21     _parentNode._decree,
22     _parentNode._entryId);
23
24     await _parentNode.learner.WriteSingleDecreeToLedgerImmediately(success);
25     await SendSuccessMessage(success);
26 }
27 _parentNode._status = NodeStatus.idle;
28 }

```

Listing 4.4: GetOutcome simply gets the only decree possible in a single-decree ledger: decree 1.

```

1 public static class LedgerHelper
2 {
3     public static async Task<byte[]> GetOutcome()
4     {
5         LedgerEntry outcomeEntry = null;
6         using (Ledger ledger = new Ledger())
7         {
8             outcomeEntry = await ledger.Entries.SingleOrDefaultAsync(e => e.Id == 1);
9         }
10
11         if (outcomeEntry == null)
12         {
13             return null;
14         }
15         else
16         {
17             return MessageHelper.StringToByteArray(outcomeEntry.Decree);
18         }
19     }
20 }

```

4.3.3.20 Proposer: Step 5 - Send Success Message

If the outcome is known, a *Success* message will be sent to all online peers.

Pseudocode

Send *Success* Message

Enabled whenever $outcome[p] \neq \text{BLANK}$.

- Send a *Success*($outcome[p]$) message to any peer.

.NET code

```

1 private async Task SendSuccessMessage(Success success)
2 {
3     byte[] outcome = await LedgerHelper.GetOutcome();
4     if (outcome != null)
5     {
6         Console.WriteLine("[Proposer] Sending passed decree [{0}:{1}] to learners.",
7             _parentNode._entryId, MessageHelper.ByteArrayToString(outcome));
8         await _parentNode.client.SendMessageToCluster(success, _parentNode.peers, true);
9     }
10 }

```

```

8     }
9     else
10    {
11        Console.WriteLine("[Proposer] Not sending success message, since outcome is not
            known.");
12    }
13 }

```

4.3.3.21 Learner: Step 6 - Receiving a Success Message

On receiving a *Success* message, a node will write the outcome to its ledger.

Pseudocode

Receive *Success*(*d*) Message

If *outcome*[*p*] = BLANK, then

- Set *outcome*[*p*] to *d*.

.NET code

```

1  public async Task ReceiveSuccess(Success success)
2  {
3      await WriteSingleDecreeToLedgerImmediately(success);
4  }
5
6  public async Task WriteSingleDecreeToLedgerImmediately(Success success)
7  {
8      LedgerEntry entry = new LedgerEntry
9      {
10         Id = success._decreeId,
11         Decree = MessageHelper.ByteArrayToString(success._decree)
12     };
13     await AddOrUpdateSingleEntry(entry);
14 }
15
16 private async Task AddOrUpdateSingleEntry(LedgerEntry entryToWrite)
17 {
18
19     Ledger ledger = new Ledger();
20     LedgerEntry entryInDb = await ledger.Entries.SingleOrDefaultAsync(e => e.Id ==
        entryToWrite.Id);
21
22     if (entryInDb == null)
23     {
24         ledger.Entries.Add(entryToWrite);
25         Console.WriteLine("[Learner] Written new decree [{0}:{1}]",
26             entryToWrite.Id,
27             entryToWrite.Decree);
28         await ledger.SaveChangesAsync();
29     }
30 }

```

4.3.3.22 Observation of the Synod Protocol implementation

Since the *Synod Protocol* has been proven in The Part-Time Parliament, the algorithm functions exactly like expected: it's a *Fault Tolerant* algorithm for reaching consensus in an asynchronous environment. The protocol works well on the built network layer – that is, no abnormalities have been observed – and since .NET has great *async* support, writing asynchronous functions for an asynchronous environment poses no problem.

Various test cases have been written based on the *Synod Protocol*, to showcase its functionality, and showcase its *Fault Tolerant* properties. The results have been documented in B.2 (Appendix B).

4.3.4 Implementing the Multi-Decree protocol

The multi-decree parliament was derived from the *Synod protocol*, keeping the consistency and progress characteristics. Additionally, instead of having single-decree ledgers, consensus can now be reached over a series of proposed decrees, and ledgers are able to contain multiple amounts of decrees. According to Lamport, going from the *Synod protocol* to the *Multi-decree protocol* required only a few simple changes to the algorithm, thus he saw no need – or he couldn't be bothered – to write a specification of the *Multi-decree Protocol*.^[5] Therefore, this implementation of this protocol is solely based on a few descriptions from the Part-Time Parliament. An overview of this description can be read in Section 3.5.4.

The description of the implementation takes the flow of a normal operation of the *Multi-decree protocol*, in which the most important changes from the *Synod Protocol* are covered. Methods which underwent minor or no changes have been left out.

4.3.4.1 Proposer: On receive leadership

The protocol starts whenever a single new president is assigned (in the code, we refer to the president as *leader*). A number of operations have to be executed before ballots can be conducted by the president. This includes (1) sending a *NewBallot*(*b*, *n*) message (where *n* is the last decree id of the decrees it has written, until it's missing decrees), (2) learning about new decrees from received *LastVote* messages, and conducting ballots for these learned decrees, and, (3) informing non-presidents with missing decrees with id's $\leq n$, and (4) filling any gaps in the ledger by conducting ballots for "olive-day" decrees. Finally, the president is able to conduct ballots for any newly proposed decrees, either from its client side, or from a different node.

.NET code

```
1 public async Task OnReceiveLeadership()
2 {
3     if (!leadershipInitTasksFinished && _parentNode._isLeader)
4     {
5         Console.WriteLine("[Proposer] Learning decrees...");
6         int newBallotMsgResult = 1;
7         await IncrementBallotId();
8
9         do
10        {
11            //execute step 1-2 to learn about decrees and prepare for steps 3-6
12            newBallotMsgResult = await TryNewBallot();
13        } while (newBallotMsgResult != 0 && _parentNode._isLeader);
14
15        if (newBallotMsgResult == 0)
16        {
```

```

17     _parentNode._status = NodeStatus.idle;
18     await ConductBallotsMissingDecreases();
19     await FillGapsInLedger();
20     leadershipInitTasksFinished = true;
21     InitExecutePaxosWhenProposerTask();
22 }
23 }
24 }

```

4.3.4.2 Proposer (president): Step 1 - Sending a NextBallot Message

Step 1 of the *Synod protocol* is to get the latest ballot id from the other nodes, so that the president can begin a ballot with a higher ballot id. If the *NextBallot*(b, n) message was successful, that ballot id will be used for all of the upcoming ballots, until a new president is elected. Value n is the decree id p has in its ledger, until it's missing any decree. (E.g. If it has decree ids 1,2,4,5 written, n would be 2, since 3 is missing.)

.NET code

```

1  public async Task<int> TryNewBallot()
2  {
3      bool majorityOnline = GetOnlineNodes().HasMajorityOf(_parentNode.allNodes);
4
5      if (_parentNode._isLeader)
6      {
7          _parentNode._status = NodeStatus.trying;
8          _parentNode._prevVotes = new List<LastVote>();
9          return await SendNextBallotMessage();
10     }
11     return 2;
12 }

```

Listing 4.5: Value n is called by calling the *GetLastEntryIdUntilMissingData()* method, and is subsequently sent with the *NextBallot*(b, n) message.

```

1  public async Task<int> SendNextBallotMessage()
2  {
3      timeAtPreviousAction = DateTime.Now;
4      Cluster setOfNodes = GetOnlineNodes();
5      bool majorityOnline = setOfNodes.HasMajorityOf(_parentNode.allNodes);
6
7      if (_parentNode._status == NodeStatus.trying
8          && _parentNode._isLeader
9          && majorityOnline)
10     {
11         Console.WriteLine("[Proposer] Sending nextBallotMsg to {0},", String.Join(",",
12             setOfNodes.Keys));
13         timeAtPreviousAction = DateTime.Now;
14         long lastEntryUntilMissing = await LedgerHelper.GetLastEntryIdUntilMissingData();
15         NextBallot nextBallot = new NextBallot(_parentNode.client._messageIdCounter,
16             _parentNode._id, _parentNode._lastTried, lastEntryUntilMissing);
17
18         //send message to peers
19         await _parentNode.client.SendMessageToCluster(nextBallot,
20             GetClusterExcludingSelf(setOfNodes), true);
21
22         //send this message to own acceptor

```

```

21     await _parentNode.acceptor.OnReceiveNextBallot(nextBallot);
22
23     //wait for reply from set of nodes
24     while (_parentNode._prevVotes.Count() != setOfNodes.Count())
25     {
26         if ((DateTime.Now - timeAtPreviousAction).TotalMilliseconds >=
27             Node.MINUTE_IN_PAXOS_TIME * 22)
28         {
29             Console.WriteLine("[Proposer] Didn't get enough lastvote messages in time.");
30             return 1;
31         }
32         Console.WriteLine("[Proposer] Received all lastvotes from chosen set of nodes.");
33         return 0;
34     }
35     else if(!majorityOnline)
36     {
37         return 1;
38     }
39     else if (!_parentNode._isLeader)
40     {
41         Console.WriteLine("[Proposer] Cannot send a NextBallot message as a non-leader.");
42     }
43     else if (_parentNode._status != NodeStatus.trying)
44     {
45         Console.WriteLine("[Proposer] Cannot send a NextBallot message when not trying.");
46     }
47     return 2;
48 }

```

Listing 4.6: This is the method which gets value n . It goes through all entries in the ledger, stops if there is a decree id missing (or stumbles upon an “olive-day decree”), and returns the previous id.

```

1  public static class LedgerHelper
2  {
3      public static async Task<long> GetLastEntryIdUntilMissingData()
4      {
5          long previousId = 0;
6          List<LedgerEntry> entries = await GetEntries();
7
8          if (entries.Count() >= 1)
9          {
10             for (int i = 0; i < entries.LastOrDefault().Id; i++)
11             {
12                 LedgerEntry entry = entries.ElementAt(i);
13
14                 if (entry.Id != previousId + 1 || entry.Decree.Equals(Proposer.OLIVE_DAY_DEGREE))
15                 {
16                     break;
17                 }
18                 else
19                 {
20                     previousId = entry.Id;
21                 }
22             }
23         }
24         return previousId;
25     }
26 }

```

4.3.4.3 Acceptor: Step 2 - Reply to NextBal(b, n) with LastVote message

When a node q receives a $NextBal(b, n)$ message with a higher ballot id than $nextBal[q]$, it will respond by sending any decrees higher than n in addition to the usual $LastVote$ information. If the ballot id is lower than $nextBal[q]$, it will instead reply with a message containing $nextBal[q]$. (Message $UpdateBallotNumber$.)

.NET code

```

1 public async Task OnReceiveNextBallot(NextBallot nextBallotMsg)
2 {
3     if (nextBallotMsg._ballotId >= _parentNode._nextBal)
4     {
5         Console.WriteLine("[Acceptor] Received new nextballot message.");
6         _parentNode._nextBal = nextBallotMsg._ballotId;
7         await SendLastVoteMessage(nextBallotMsg);
8         await LedgerHelper.SavePaxosProgressAsync(_parentNode);
9     }
10    else if (nextBallotMsg._ballotId < _parentNode._nextBal)
11    {
12        Console.WriteLine("[Acceptor] Received old nextballot message.");
13        await ReplyNextBal(nextBallotMsg._senderId);
14    }
15 }

```

```

1 private async Task ReplyNextBal(int peerId)
2 {
3     Console.WriteLine("[Acceptor] Sending nextBal to president {0}.", peerId);
4     UpdateBallotNumber newBallot = new
5         UpdateBallotNumber(_parentNode.client._messageIdCounter, _parentNode._id,
6         _parentNode._nextBal);
7     await _parentNode.client.SendMessageToNode(newBallot, peerId, true, true);
8 }

```

Listing 4.7: Upon sending a $LastVote$ message (containing p 's missing decrees), node q will send a separate message, asking for any decrees less than or equal to n in p 's ledger.

```

1 private async Task SendLastVoteMessage(NextBallot nextBallotMsg)
2 {
3     if (_parentNode._nextBal > _parentNode._prevBal)
4     {
5         string missingDecreasesPresidentString
6         = nextBallotMsg._senderId == _parentNode._id
7         ? ""
8         : await LedgerHelper.GetMissingEntriesPresident(_parentNode._id,
9         nextBallotMsg._senderId,
10        nextBallotMsg._hasDecreasesUntil);
11
12        LastVote lastVote = new LastVote(_parentNode.client._messageIdCounter,
13        _parentNode._nextBal,
14        _parentNode._id,
15        _parentNode._prevBal,
16        _parentNode._prevDec,
17        missingDecreasesPresidentString);
18
19        if (nextBallotMsg._senderId == _parentNode._id)
20        {
21            Console.WriteLine("[Acceptor] Sending LastVote to self");

```

```

22     _parentNode.proposer.ReceiveLastVoteMessage(lastVote);
23 }
24 else
25 {
26     Console.WriteLine("[Acceptor] Sending LastVote to president (node {0})",
27         nextBallotMsg._senderId);
28     Node votingProposer = _parentNode.onlinePeers.GetNodeById(nextBallotMsg._senderId);
29     await _parentNode.client.SendMessageToNode(lastVote, votingProposer, true, true);
30     await RequestMissingEntries(nextBallotMsg);
31 }
32 }
33 else
34 {
35     Console.WriteLine("[Acceptor] Not sending lastvote because prevBal({0}) is higher
36         than nextBal({1})", _parentNode._nextBal, _parentNode._prevBal);
37 }
38 }

```

Listing 4.8: Creates a string, containing all entries (id:decree) higher than n , missing in president p 's ledger.

```

1 public static async Task<string> GetMissingEntriesPresident(int parentNodeId, int
2     requestingNodeId, long decreeId)
3 {
4     string missingEntriesString = "";
5     List<LedgerEntry> entriesToInformLeader = new List<LedgerEntry>();
6
7     using (Ledger ledger = new Ledger())
8     {
9         entriesToInformLeader = await
10             ledger.Entries
11                 .Where(e => e.Id > decreeId)
12                 .Where(e => !e.Decree.Equals(Proposer.OLIVE_DAY_DEGREE))
13                 .ToListAsync();
14     }
15
16     if (entriesToInformLeader.Count() > 0)
17     {
18         for (int i = 0; i < entriesToInformLeader.Count(); i++)
19         {
20             missingEntriesString += String.Format("{0}:{1}",
21                 entriesToInformLeader.ElementAt(i).Id,
22                 entriesToInformLeader.ElementAt(i).Decree);
23             if (i < entriesToInformLeader.Count() - 1)
24             {
25                 missingEntriesString += "|";
26             }
27         }
28     }
29
30     return missingEntriesString;
31 }

```

4.3.4.4 Proposer (president): Send missing decrees to node q

When a node q replied to a *LastVote* message, it sent a *RequestMissingEntries* Message along with it. This message contains the decrees q has, and president p will gather all the decrees $\leq n$, compare q 's decrees with p 's decrees, filter out all the decrees q already has, and send these

decrees back to q with a *InformMissingEntries* message. This message contains a formatted string of all of q 's missing decrees.

.NET code

```

1 public async Task InformMissingDecreases(RequestMissingEntriesMessage rmem)
2 {
3     List<long> containingDecreeIds =
4     rmem._entriesInOwnLedgerString.Length > 0
5     ? rmem._entriesInOwnLedgerString.Split('|').Select(d => long.Parse(d)).ToList()
6     : new List<long>();
7
8     string entriesToInformString = await
9         LedgerHelper.GetMissingEntriesForNonPresident(rmem._decreeId, containingDecreeIds);
10
11     InformMissingEntriesMessage informMissingEntriesMessage =
12     new InformMissingEntriesMessage(_parentNode.client._messageIdCounter,
13     _parentNode._id,
14     entriesToInformString);
15
16     if (entriesToInformString.Length > 0)
17     {
18         Console.WriteLine("[Proposer] Informing {0} with missing decrees: [{1}]",
19             rmem._senderId, entriesToInformString);
20         await _parentNode.client.SendMessageToNode(informMissingEntriesMessage,
21             rmem._senderId, true, true);
22     }
23 }

```

4.3.4.5 Learner (non-president): Learn about missing decrees

After having received the *InformMissingDecreases* message from president p , q will parse the message containing all of its missing decrees, and will write these decrees to its ledger.

.NET code

Listing 4.9: The decree string parsing method. After having parsed all of the decrees, it will call a method to write the decrees to the ledger.

```

1 public async Task WriteMissingDecreasesToLedger(string missingDecreasesString)
2 {
3     //learn about any missing decrees sent by lastvote from other priests
4     if (missingDecreasesString.Length > 0)
5     {
6         Console.WriteLine("[Learner] Writing missing entries to ledger.");
7         string[] missingDecreases = missingDecreasesString.Split('|');
8         List<LedgerEntry> ledgerEntriesToWrite = new List<LedgerEntry>();
9
10        foreach (var missingDecree in missingDecreases)
11        {
12            string[] entryInformation = missingDecree.Split(':');
13            LedgerEntry ledgerEntry = new LedgerEntry
14            {
15                Id = long.Parse(entryInformation[0]),
16                Decree = entryInformation[1]
17            };
18            ledgerEntriesToWrite.Add(ledgerEntry);
19        }
20        await AddOrUpdateLedgerEntries(ledgerEntriesToWrite);

```



```

21     }
22 }

```

Listing 4.10: The method for writing multiple entries to the ledger.

```

1 private async Task AddOrUpdateLedgerEntries(List<LedgerEntry> entriesToWrite)
2 {
3     using (Ledger ledger = new Ledger())
4     {
5         foreach (LedgerEntry entryToWrite in entriesToWrite)
6         {
7             await AddOrUpdateSingleEntry(entryToWrite, ledger);
8         }
9         await ledger.SaveChangesAsync();
10    }
11 }

```

Listing 4.11: Method for writing a single ledger (or updating if it's an olive-day decree.)

```

1 private async Task AddOrUpdateSingleEntry(LedgerEntry entryToWrite, Ledger ledger)
2 {
3     bool doSave = false;
4     if (ledger == null)
5     {
6         doSave = true;
7         ledger = new Ledger();
8     }
9
10    LedgerEntry entryInDb = await ledger.Entries.SingleOrDefaultAsync(e => e.Id ==
11        entryToWrite.Id);
12
13    if (entryInDb == null)
14    {
15        ledger.Entries.Add(entryToWrite);
16        Console.WriteLine("[Learner] Written new decree [{0}:{1}]",
17            entryToWrite.Id,
18            entryToWrite.Decree);
19    }
20    else if (entryInDb.Decree.Equals(Proposer.OLIVE_DAY_DEGREE))
21    {
22        entryInDb.Decree = entryToWrite.Decree;
23        Console.WriteLine("[Learner] Updated decree [{0}:{1}]",
24            entryInDb.Id,
25            entryInDb.Decree);
26    }
27
28    if (doSave)
29    {
30        await ledger.SaveChangesAsync();
31    }
32 }

```

4.3.4.6 Proposer (president) - step 3: Conduct ballots for missing decrees

When the president received all *LastVote* messages from a group of nodes Q , it will filter out (1) the decrees it already has from the requested decrees sent by Q (2) duplicate decrees among the decrees sent by Q . Then, it will attempt to begin balloting for these decrees, to try to get them written in the ledgers.

.NET code

Listing 4.12: The method for weeding out duplicate decrees, and conducting ballots for learned decrees.

```

1 public async Task ConductBallotsMissingDecrees()
2 {
3     //learn about any missing decrees sent by lastvote from other priests
4     if (_parentNode._isLeader && _parentNode._prevVotes.Count() >= 1)
5     {
6         List<LedgerEntry> entries = await LedgerHelper.GetEntries();
7         HashSet<long> decreesInLedger = new HashSet<long>();
8
9         List<string> missingDecreesStrings =
10        _parentNode._prevVotes.Select(v => v._missingDecrees)
11        .Where(s => s.Length > 0).ToList();
12
13        Dictionary<long, byte[]> decreesToPropose = new Dictionary<long, byte[]>();
14
15        foreach (string missingDecreesString in missingDecreesStrings)
16        {
17            string[] missingDecrees = missingDecreesString.Split('|');
18
19            foreach (var missingDecree in missingDecrees)
20            {
21                string[] entryInformation = missingDecree.Split(':');
22                long entryId = long.Parse(entryInformation[0]);
23                byte[] decree = MessageHelper.StringToByteArray(entryInformation[1]);
24
25                //if not sure in ledger
26                if (!decreesInLedger.Contains(entryId))
27                {
28                    LedgerEntry entryInDb = entries.SingleOrDefault(e => e.Id == entryId);
29                    if (entryInDb != null && !entryInDb.Decree.Equals(OLIVE_DAY_DEGREE))
30                    {
31                        //just discovered is in ledger
32                        //also make sure to override any olive day decrees
33                        decreesInLedger.Add(entryId);
34                        continue;
35                    }
36                }
37                else// if decree known in ledger
38                {
39                    continue;
40                }
41
42                if (!decreesToPropose.ContainsKey(entryId))
43                {
44                    decreesToPropose.Add(entryId, decree);
45                }
46            }
47        }
48
49        foreach (var decreeToPropose in decreesToPropose)
50        {
51            //not in ledger, so needs to be proposed/written
52            Console.WriteLine("\n[Proposer] Learned about [{0}:{1}]. Conducting ballot.",
53            decreeToPropose.Key,
54            MessageHelper.ByteArrayToString(decreeToPropose.Value));
55            _parentNode._entryId = decreeToPropose.Key;
56            _parentNode._decree = decreeToPropose.Value;

```

```

57     await ExecutePaxos(decreeToPropose.Value, false, false, decreeToPropose.Key);
58 }
59 }
60 }

```

4.3.4.7 Proposer (president) - step 3: Conducting a ballot

The method for starting a round of Paxos was expanded with more code. Some adaptations were made such as passing various variables (see the arguments of the method), and extra checks were added such as executing the algorithm under certain circumstances. Retrying sending messages was reworked by returning a code to see if it was successful, instead of dangerously using recursive calls until success.

The `StartPollingMajoritySet` method has been changed a lot to fit the criteria of the multi-decree parliament; it has to be compatible with (1) old/learned decrees, (2) filling gaps with unimportant decrees, and (3) newly proposed decrees. The decree id is determined from these variables, and the decree is determined by requirement 3 (preliminary protocol) if it's for old/learned decrees. Otherwise, the ballot's decree can be any decree. One with the current implementation of this method, is that the parliament does not expect the quorum members to leave during balloting. Based on the precise description (pseudocode of the *basic protocol*) this is especially problematic during balloting for newly proposed decrees, since this leads to a stop of the execution. A solution could be to not require all of the quorum members to be voters during balloting for newly proposed decrees, however there is not yet a tested solution found. A test case was written for this problem in B.3.9

.NET code

```

1  public async Task ExecutePaxos(byte[] proposedDecree, bool isFill = false, bool
    isNewDecree = false, long entryId = 0)
2  {
3      Cluster quorum = GetOnlineNodes();
4      bool initFinished = ((!isNewDecree && !leadershipInitTasksFinished) ||
    leadershipInitTasksFinished);
5
6      if (_parentNode._isLeader
7      && _parentNode._status == NodeStatus.idle
8      && initFinished)
9      {
10         Console.WriteLine("\n[Proposer] Executing Paxos");
11
12         int ballotSuccessful = 1;
13         do
14         {
15             quorum = GetOnlineNodes();
16             _parentNode._status = NodeStatus.trying;
17             ballotSuccessful = await StartPollingMajoritySet(quorum, proposedDecree, isFill,
    isNewDecree, entryId);
18         } while (ballotSuccessful == 1);
19
20         if (ballotSuccessful == 0)
21         {
22             await Succeed();
23         }
24         else if (ballotSuccessful == 2)
25         {
26             Console.WriteLine("[Proposer] Aborting Paxos");
27         }

```

```

28     }
29     else if (!initFinished)
30     {
31         Console.WriteLine("[Proposer] Not ready to conduct ballot.");
32     }
33 }

```

```

1     private async Task<int> StartPollingMajoritySet(Cluster quorum, byte[] proposedDecree,
2         bool isFill, bool isNewDecree, long entryId = 0)
3     {
4         bool quorumMembersAreLastVoters =
5         quorum.Keys.ToList()
6         .Intersect(_parentNode._prevVotes.Select(lv => lv._senderId).ToList())
7         .Count() == quorum.Count();
8
9         if (_parentNode._isLeader
10        && _parentNode._status == NodeStatus.trying
11        && quorumMembersAreLastVoters)
12        {
13            Console.WriteLine("[Proposer] Polling...");
14            _parentNode._status = NodeStatus.polling;
15            _parentNode._quorum = quorum;
16            _parentNode._voters = new Cluster();
17
18            _parentNode._isFill = isFill;
19            _parentNode._isNewDecree = isNewDecree;
20
21            if (entryId != 0) //if entryId has been given
22            {
23                _parentNode._entryId = entryId;
24            }
25            else //if entryId has not been given
26            {
27                if (!isFill) //when it's a normal entry
28                {
29                    LedgerEntry lastEntryInDb = null;
30
31                    using (Ledger ledger = new Ledger())
32                    {
33                        lastEntryInDb = await ledger.Entries.LastOrDefaultAsync();
34                    }
35
36                    if (lastEntryInDb != null)
37                    {
38                        _parentNode._entryId = lastEntryInDb.Id;
39                        _parentNode._entryId++;
40                    }
41                    else
42                    {
43                        _parentNode._entryId = 1;
44                    }
45
46                    if (isNewDecree)
47                    {
48                        _parentNode._decree = proposedDecree;
49                    }
50                    else
51                    {
52                        _parentNode._decree = GetDecreeToPropose(proposedDecree);
53                    }

```

```

53     }
54     else //when it's filling with olive day decrees
55     {
56         _parentNode._decree = proposedDecree;
57     }
58 }
59
60 Console.WriteLine("[Proposer] Ballot: decreeId={0}, decree={1}, quorum={2}",
        _parentNode._entryId, MessageHelper.ByteArrayToString(_parentNode._decree),
        String.Join(",", quorum.Keys.ToArray()));
61 return await SendBeginBallotMessage();
62 }
63 return 1;
64 }

```

4.3.4.8 Proposer (president) - Step 5: Succeed + send success message

The *GetOutcome()* method has been adapted to accept a decree id as a parameter, so that a check can be done to see if the ledger contains the passed decree.

.NET code

```

1 private async Task Succeed()
2 {
3     timeAtPreviousAction = DateTime.Now;
4     bool quorumMembersAreVoters =
5     _parentNode._quorum.Keys.ToList()
6     .Intersect(_parentNode._voters.Keys.ToList())
7     .Count() == _parentNode._quorum.Count();
8
9
10    byte[] outcome = await LedgerHelper.GetOutcome(_parentNode._entryId);
11
12    if (_parentNode._status == NodeStatus.polling
13    && quorumMembersAreVoters
14    && outcome == null || (outcome != null &&
15        MessageHelper.ByteArrayToString(outcome).Equals(OLIVE_DAY_DEGREE)))
16    {
17        Console.WriteLine("[Proposer] Ballot [{0}:{1}] succeeded.",
18            _parentNode._entryId,
19            MessageHelper.ByteArrayToString(_parentNode._decree));
20
21        Success success = new Success(_parentNode.client._messageIdCounter,
22            _parentNode._id,
23            _parentNode._decree,
24            _parentNode._entryId);
25
26        await _parentNode.learner.WriteSingleDecreeToLedgerImmediately(success);
27        await SendSuccessMessage(success);
28    }
29    else if (!quorumMembersAreVoters)
30    {
31        Console.WriteLine("[Proposer] Not succeeding ballot. Not all quorum members voted.");
32    }
33    else if (_parentNode._status != NodeStatus.polling)
34    {
35        Console.WriteLine("[Proposer] Not succeeding ballot, Node is not polling.");
36    }
37    else if (outcome != null)

```

```

37     {
38         Console.WriteLine("[Proposer] Not succeeding ballot, Outcome is already known: {0}",
39                             MessageHelper.ByteArrayToString(outcome));
40     }
41     _parentNode._status = NodeStatus.idle;
42 }

```

```

1 private async Task SendSuccessMessage(Success success)
2 {
3     byte[] outcome = await LedgerHelper.GetOutcome(_parentNode._entryId);
4     if (outcome != null)
5     {
6         Console.WriteLine("[Proposer] Sending passed decree [{0}:{1}] to
7                             learners.",
8                             _parentNode._entryId,
9                             MessageHelper.ByteArrayToString(outcome));
10        await _parentNode.client.SendMessageToCluster(success, _parentNode.peers, true);
11    }
12    else
13    {
14        Console.WriteLine("[Proposer] Not sending success message, since outcome is not
15                            known.");
16    }
17 }

```

Listing 4.13: The updated GetOutcome function, which is now able to get the outcome of not just the first (and only) decree, but any given decree id.

```

1 public static class LedgerHelper
2 {
3     public static async Task<byte[]> GetOutcome(long decreeId)
4     {
5         LedgerEntry outcomeEntry = null;
6         using (Ledger ledger = new Ledger())
7         {
8             outcomeEntry = await ledger.Entries.FindAsync(decreeId);
9         }
10
11        if (outcomeEntry == null)
12        {
13            return null;
14        }
15        else
16        {
17            return MessageHelper.StringToByteArray(outcomeEntry.Decree);
18        }
19    }
20 }

```

4.3.4.9 Learner - Step 5/6: Write decree to ledger

The president executes this code in step 5, while a non-president acceptor executes this code in step 6 upon receiving the *Success* message. The *Success* message was expanded to also contain the decree id. The learners can thus easily get the passed decree information from the *Success* message.

.NET code

```

1 public async Task WriteSingleDecreeToLedgerImmediately(Success success)
2 {
3     LedgerEntry entry = new LedgerEntry
4     {
5         Id = success._decreeId,
6         Decree = MessageHelper.ByteArrayToString(success._decree)
7     };
8     await AddOrUpdateSingleEntry(entry, null);
9 }

```

4.3.4.10 Proposer (president): Fill gaps in ledger with olive-day decrees

The president creates a list of entry id's missing in its ledger, and starts balloting with the decree being an “olive-day” decree for each of the missing ledger entries. Once balloting is successful, it will write these unimportant decrees to its ledger and send *Success* messages to its peers.

.NET code

```

1 public async Task FillGapsInLedger()
2 {
3     List<int> missingEntryIds;
4
5     if (_parentNode._isLeader)
6     {
7         List<LedgerEntry> entries = await LedgerHelper.GetEntries();
8         List<int> writtenEntryIds = entries.Select(e => (int)e.Id).ToList();
9
10        if (writtenEntryIds.Any())
11        {
12            missingEntryIds = Enumerable.Range(1,
13                (int)writtenEntryIds.Last()).Except(writtenEntryIds).ToList();
14
15            foreach (int entryId in missingEntryIds)
16            {
17                if (_parentNode._isLeader)
18                {
19                    Console.WriteLine("\n[Proposer] Attempting to fill decree {0} with olive day
20                        decree.", entryId);
21                    _parentNode._entryId = entryId;
22                    _parentNode._decree = MessageHelper.StringToByteArray(OLIVE_DAY_DEGREE);
23                    await ExecutePaxos(_parentNode._decree, true, false, entryId);
24                }
25                else
26                {
27                    return;
28                }
29            }
30
31            if (missingEntryIds.Count() > 0)
32            {
33                Console.WriteLine("[Proposer] Filled gaps in ledger with olive day decree.");
34            }
35        }
36        else
37        {
38            Console.WriteLine("[Proposer] No gaps to fill with olive day decrees.");
39        }
40    }

```

38	}
39	}

4.3.4.11 Observation of the Multi-decree Protocol implementation

The *Multi-Decree* protocol allows – as claimed – a series of decrees to be written to the ledgers. It works as expected, however there are two issues: (1) if multiple *Success* messages are rapidly sent to the non-president nodes’ Learner, occasionally *Success* messages are skipped, and these decrees will not be written in the ledger. (2) The *Multi-decree* protocol expects a static quorum—no node should join it or leave it. This seems to be the opposite of *Fault Tolerant* behavior, so the implementation as it is, is less *Fault Tolerant*. Nonetheless, if these two issues can be solved, the *Multi-decree protocol* could be a valuable consensus algorithm for a private blockchain.

Various tests have been written based on the *Multi-decree protocol* as well. The results have been documented in B.3, including problem 1 (B.3.7) and problem 2 (B.3.9).

4.4 Recap

The development process of the The Paxos replication has been covered in detail, and we have seen how the Paxos implementation was developed in order to meet its requirements.

The written network layer – using a reinforced UDP protocol on application layer – seems to be capable of reliably sending packages across a peer-to-peer network (see test case B.1.2). Pure implementations (that is, made exactly as described in The Part-Time Parliament) of both the *Synod protocol* and the *Multi-decree protocol* have been made on top of this network layer. The *Synod Protocol* allows a majority of nodes to reach consensus over a single value in a *Fault Tolerant, asynchronous* environment. The *Multi-Decree protocol* fulfills the requirement of writing a series of values, but the implementation⁴ has one flaw: it assumes a static number of participants[5, 16], and alas the application comes to a halt if a node joins *after* the president has finished its preparation for receiving proposals. A solution must be found for this one weakness, since this damages its *Fault Tolerance*, one of its vital characteristics.

Immutability of the written decrees has been achieved by adding triggers on *insert* and *update* actions on Database level, but does not stop users from inserting new entries and editing olive-day decrees by hand. A possible solution might be to restrict user permissions on the database.

Multiple tests have been executed and documented in Appendix B. These tests prove the functionality and characteristics of the written application, and shows that it meets the Paxos requirements described in The Part-Time Parliament.

⁴It doesn’t have to be the implementation necessarily, it might as well be algorithm itself. It is unsure which of the two, due to the missing specification of the *Multi-decree protocol*.

Chapter 5

Conclusion

In this research we tried to get an answer to the question: “*What value does the Paxos algorithm have for reaching consensus in the context of a blockchain?*”. To answer this question, we’ve conducted a case study of the article *The Part-Time Parliament*, which describes the Paxos consensus algorithm in detail. During research, we’ve applied the research’s findings to create an implementation of the *Synod protocol* and the *Multi-decree protocol* of the Paxos algorithm, described in *The Part-Time Parliament*.

Based on the results, we have seen that the *Synod protocol* is a highly effective *Fault Tolerant* consensus algorithm for reaching consensus in an asynchronous, unreliable network of nodes. The biggest limitation to the *Synod* protocol, is that it only allows to reach consensus over one single, final value. On the other hand, the *Multi-Decree protocol* is capable of reaching consensus about a series of values for a distributed ledger, and provides the same qualities as the *Synod protocol*. However, the built implementation of the *Multi-Decree protocol* comes to a halt if new nodes join the process of reaching consensus, after the leading node has done its preparation for proposing and writing a series of values. A solution has to be found for this single issue, since it slightly weakens its *Fault Tolerance* property. Regardless of deadlocks, consistency will remain.

Immutability was achieved by implementing triggers on insert and update commands on database level. These triggers prevent users from changing the contents of already written values, although this does not stop users from inserting new values by hand, or updating unimportant values with a different value. Paxos does not contain anything to combat users from editing their database on database level, even though it’s a vital requirement. Besides, a pure implementation of Paxos is not Byzantine Fault Tolerant by itself. Thus, to meet that characteristics of blockchain, an additional method for handling the Byzantine generals problem must be researched and applied to achieve Byzantine Fault Tolerance.

From this applied research it appeared that the *Multi-Decree* Paxos algorithm described in *The Part-Time Parliament* is indeed a valuable consensus algorithm for blockchains, provided a solution has been found for the shortcoming of the implementation described, and assuming the algorithm is reinforced with (1) Byzantine Fault Tolerance, (2) guarantees immutability of the data, and (3) restricts users from inserting or updating data on database level.

Chapter 6

Discussion

A case study was conducted on *The Part-Time Parliament* for a complete understanding of the Paxos algorithm. An additional article called *Paxos Made Simple* was used for understanding the Paxos algorithm in a more simple way, and was used whenever ideas or concepts discussed in *The Part-Time Parliament* were unclear. Based on these two articles, an implementation of the Paxos algorithm was written in .NET core. The assumption is that if the same applied research is conducted, similar to equal results will be achieved regardless of the programming language or environment, since the protocols don't change.

From the results we have concluded that the Paxos protocol is a valuable *fault tolerant* consensus algorithm for reaching consensus on a closed environment (private blockchain), as long as the blockchain implements various patches to meet the requirements of blockchain: (1) Set database triggers on update and delete commands, and make the user unable to edit the ledgers by hand (2) Implement Byzantine Fault Tolerance, since Paxos can't deal with Byzantine Failures (3) Don't require the quorum of the *Multi-decree protocol* to be the same set of nodes (or less nodes of the same set) after preparation of the *president*—new nodes joining after preparation is a problem with the written implementation, since it is assumed in the *Multi-decree protocol* that no node goes offline or online[5, 16].

The first and second improvements were expected—database settings and permissions probably needs to be set on database level (or it could be solved using a Merkle tree), and it was already stated early in *The Part-Time Parliament* (although not explicitly) that the Paxon Parliament didn't know how to deal with Byzantine Failures. Contrary to the first and second improvement, the third improvement was unexpected, because it was assumed that the Paxos protocol by itself was proven, documented, and fully Fault Tolerant. However, Lamport only proved the functionality of the *Synod Protocol* in *The Part-Time Parliament*. Therefore a possible explanation of this issue might be that Lamport overlooked this one problem, or a mistake has been made in the implementation of the applied research.

Another problem with the current implementation is that it can't deal with a high amount of *Success* messages, due to the creation of the *Ledger* class and continuously saving these changes on each received *Success* message. A solution could be to save multiple successful decrees at once after a specific time-out (for non-presidents. Presidents have to write the decree immediately after finding out the ballot was successful). Less instances of *Ledger* will be created, and as a result less saves will be done on the database, which should increase the performance (in theory). This is a slightly improved concept of the adaptation mentioned in B.3.7. It's possible that the chosen Object-relational mapping *Entity Framework Core* causes this delay, and using a different ORM would not cause any performance issues on saving to the ledger.

This research was conducted based on a general research behind the characteristics of blockchain, which briefly mentioned the Paxos algorithm as an example for reaching consensus on a distributed network.[1] Thus, the aim of this research was to understand the value of the Paxos algorithm (described in *The Part-Time Parliament*) in the context of a blockchain. The results show that it is possible to reach consensus about an indefinite amount of values. The implementation was restricted so that the decrees written in the ledgers solely consist of strings

– that is, pieces of text. In *The Part-Time Parliament*, the Paxos algorithm is applied to a distributed state-machine, which in actuality could be of real use for a blockchain. Besides, Leslie Lamport wrote many more articles on Paxos, which mostly describe variations and adaptations of the original Paxos algorithm. These algorithms could prove to be even more beneficial for blockchain.

Therefore, it is recommended that future research finds solutions to the improvements on Paxos mentioned earlier, and applies a state-machine approach to the Paxos algorithm. Secondly, Lamport mentions more ways of expanding the *Multi-decree protocol* in *The Part-Time Parliament*. These might be worth looking at, even though it's not vital for correct functioning of the algorithm. Thirdly, further research could be conducted on the usefulness of the variations of Paxos, described in articles such as *Disk Paxos*[14], *Fast Paxos*[15] and more written by Leslie Lamport. Lastly, since the CoCo framework will contain Paxos – and since it still has yet to be released – it might be worth comparing the made implementation with the Paxos implementation of the CoCo framework, once the source code comes available.

Chapter 7

Reflection

7.1 Teacher's concerns

At the start the mandate leading to this project had not been accepted. This was due to the first university teacher's concerns; he thought the application would be too ambitious to complete successfully. Fortunately, after multiple visits at the Rotterdam University of Applied Sciences and having shown a prototype of the network layer, my first University teacher was more confident in seeing what was built, and accepted my mandate.

7.2 Understanding the Part-Time Parliament

Lots of time was spent on reading the Part-Time parliament. It was a difficult paper to get through, due to its metaphor, and due to its density of information. It took a very long time – together with the other interns – to understand just the gist of the document. External resources, documents, videos, articles, were not permitted since the supervisor wanted a pure implementation of the Paxos algorithm, and external sources could potentially disturb the understanding of the algorithm. Coupled with the facts that only few people really understand the article, and that the article was hard to decipher due to the metaphor used, it took a very long time to understand what the algorithm does. Nonetheless, this was a great opportunity to learn how to analyze an article, break it down to smaller pieces, and apply it directly to a project.

7.3 Development environment

Developing a peer-to-peer application requires multiple computers to test the application on. Using docker or virtual machines was an option, but those would add extra unneeded and unwanted complexity. I spent a lot of time working from home, since I had no different environment to develop and test the application. Finally, after months of waiting I finally got multiple Ubuntu servers from Centric to deploy my application.

7.4 Illness

For the entire duration during my internship I have dealt with chronic physical pain. Unfortunately this made it really tough to concentrate for longer periods of time, and as a result was researching a complex topic such as Paxos even more difficult and time-consuming.

7.5 Validation

Validation of the developed product felt impossible at times, since I had no one to go to if I wanted to know something, except for two other interns who did some global research to process Paxos into their research. So thankfully I wasn't entirely on my own, but a lot of the time it felt like I had to figure out everything by myself. (Of course, you could see this as an important step towards independence.) Nonetheless, I didn't have anyone or anything to verify my application (except for *The Part-Time Parliament* and *Paxos Made Simple*), so often I was unsure if I was building the application the way it is supposed to be built.

7.6 Ambition

At the end of the project (around week 17 of 20) I had finished the *Synod Protocol*, but I felt the desire to achieve more; I wanted to finish the *Multi-Decree protocol*. I thought this was easily achievable, however on my way to victory there were many unexpected obstacles. The deadline kept coming closer and closer while I felt like all I was doing was fixing bugs. It was a very stressful period, however ultimately I'm glad I tried to give it everything I had, and managed to still deliver a complete product.

7.7 Overall thoughts

Even though not everything went as expected, I'm very content with the end result. I've made something very unique, and something which not many people get the opportunity of building. It was a very educational experience for me: I learned a lot about blockchain, learned how to analyze scientific papers, wrote a complex peer-to-peer application, learned a lot about consensus algorithms, internet protocols, and much more. Ultimately, I think the biggest lesson for me was the weight of carrying such a great responsibility, and being held accountable for a project which multiple parties depend on. That responsibility helped me immensely in improving myself and my professional skills.

Appendix A

Synod protocol class diagram

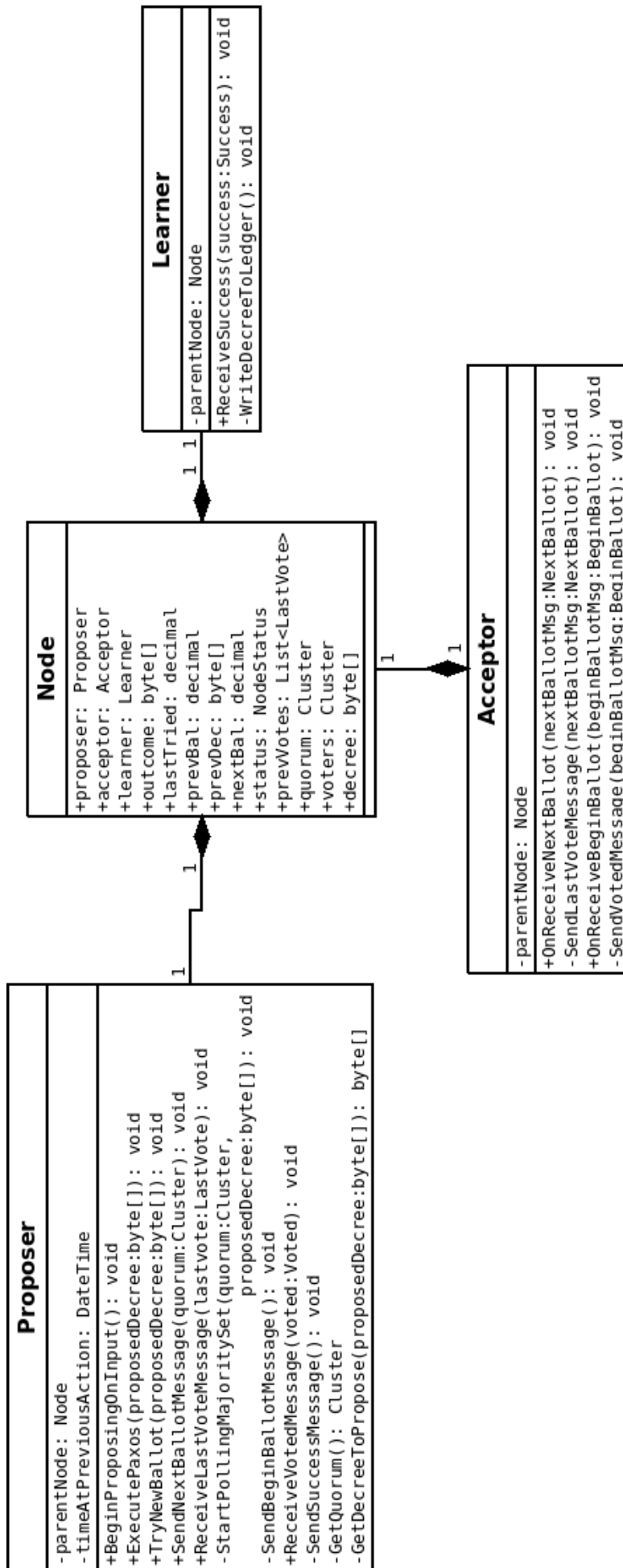


Figure A.1: Class diagram of the Synod Protocol.

Appendix B

Test results

In this chapter are various test cases of the Proof of Concept blockchain simulation. Lines starting with “>” indicate user input on the command line.

B.1 General test results

Irregardless of functionality, a number of tests apply to both versions of the protocol. These will be documented here.

B.1.1 Immutability

In chapter 4.3.2.2 we mentioned that triggers on the UPDATE and DELETE commands would make important decrees immutable. Here we are putting it to the test by attempting these actions on database level. As you can see, updating and deleting important decrees is not allowed, and returns an error code. However, inserting decrees and editing unimportant decrees by hand can still be done. A possible solution might be setting user permissions.

```
sqlite> select * from Entries;
1|Test
sqlite> DELETE FROM Entries WHERE Id = 1;
Error: Ledgers are written with indelible ink, and their entries can not be changed
sqlite> UPDATE Entries SET Decree="Test2" WHERE Id = 1;
Error: Ledgers are written with indelible ink, and their entries can not be changed
sqlite> INSERT INTO Entries VALUES (2, "Test2");
sqlite> INSERT INTO Entries VALUES (3, "Test3");
sqlite> INSERT INTO Entries VALUES (4, "The ides of February is national olive day");
sqlite> UPDATE Entries SET Decree="Test4" WHERE Id = 4;
sqlite> select * from Entries;
1|Test
2|Test2
3|Test3
4|Test4
```

B.1.2 Testing the network

Nodes would be balloting for multiple decrees in the *Multi-decree protocol*, and would track how many packages were confirmed. Specifically, we are taking a look at Node 1, who would stop the application after a certain amount of messages were sent (in this case 1000), and would tell how many packages were left unconfirmed and had to be discarded. As you can see, from all of the important messages, not a single package was lost.

Node 1

```
Packages lost: 0/1000
```


B.1.3 Excluding a node

Node 3 has been removed from every node's peer file. In theory, node 3 shouldn't be able to be part of the network anymore.

Node 1

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
> Test
[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=Test, quorum=2,1
[Acceptor] Received beginballot [6.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 6.1.
[Proposer] Received voted message from 2 for 6.1.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 2

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [6.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test]
```

Node 3

```
Local IPv4 couldn't be found in list of authorized nodes.
```

B.1.4 Corrupt message

I added a function for node 1 to scramble the package bytes before sending the package. Here is shown how Node 2 handles these corrupt messages. Consistency is remained, because nothing is done with these corrupt packages.

Node 1

```
> Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
...
```

B.2.2 Simple majority formed

while node 1 is still retrying its proposal endlessly, Node 2 joins the network, and so a majority could finally be formed (2/3 nodes). Node 1 stays the president, and node 2 participates in node 1's ballot. Finally, the initial proposed decree is written in the ledger.

Node 1

```
...

[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Executing Paxos for decree=Test
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Didn't get enough lastvote messages in time.

[Proposer] Executing Paxos for decree=Test
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [172.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 172.1.
[Proposer] Received voted message from 2 for 172.1.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 2

```
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to president (node 1)  
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to president (node 1)  
[Acceptor] Received beginballot [172.1] from 1  
[Acceptor] Voting for ballot...  
[Acceptor] Sending voted to 1  
[Learner] Written new decree [1:Test]
```

B.2.3 President goes offline before sending BeginBallot message

A promise to vote for a specific decree happens with a *LastVote* message. Thus, if balloting is interrupted before that, an entirely different decree is able to be written in the ledger with a new ballot.

Node 1 (president)

```
> Test  
  
[Proposer] Executing Paxos  
[Proposer] Sending nextBallotMsg to 2,3,1,  
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to self  
[Proposer] Received lastvote from node 1.  
[Proposer] Received lastvote from node 2.  
(Exit)
```

Node 2

```
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to president (node 1)  
[Server] Received leadership.  
Received a new decree proposal for [Test2]  
  
[Proposer] Executing Paxos  
[Proposer] Sending nextBallotMsg to 3,2,  
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to self  
[Proposer] Received lastvote from node 2.  
[Proposer] Received lastvote from node 3.  
[Proposer] Received all lastvotes from chosen set of nodes.  
[Proposer] Polling...  
[Acceptor] Received beginballot [2.2] from 2  
[Acceptor] Voting for ballot...  
[Acceptor] Sending voted to self  
[Proposer] Received voted message from 2 for 2.2.  
[Proposer] Received voted message from 3 for 2.2.  
[Proposer] Ballot [1:Test2] succeeded.  
[Learner] Written new decree [1:Test2]  
[Proposer] Sending passed decree [1:Test2] to learners.
```

Node 3

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
> Test 2
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Received beginballot [2,2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [1:Test2]
```

B.2.4 President goes offline before sending Success message

Even though the president (node 1) failed during balloting, requirement 3 of the *preliminary protocol* ensures that decree “Test1” will be chosen as decree instead of “Test2”, because all nodes previously voted for decree “Test1”. The consistency requirement has therefore been met, since once a value has been voted for, that will ultimately be the final chosen value.

Node 1 (president)

```
> Test1
[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [2.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 2.1.
[Proposer] Received voted message from 2 for 2.1.
[Proposer] Received voted message from 3 for 2.1.
(Exit)
```

Node 2

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [2.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Server] Received leadership.
> Test2
[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 3,2,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
```

```
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
Success called
[Proposer] Ballot [1:Test1] succeeded.
[Learner] Written new decree [1:Test1]
[Proposer] Sending passed decree [1:Test1] to learners.
```

Node 3

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Received beginballot [2,2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [1:Test1]
```

B.2.5 Non-president goes offline before sending LastVote message

Here is shown how the *Synod protocol*'s requirement of retrying ballots after a period of time ensures progress. Node 1 failed to come to consensus twice, because it still considered Node 3 to be part of the quorum, which was already offline. With balloting attempt #3 the quorum changed to online nodes 1 and 2, and the ballot completed successfully.

Node 1

```
> Test

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Didn't get enough lastvote messages in time.

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Didn't get enough lastvote messages in time.

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
```

```
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 4.1.
[Proposer] Received voted message from 2 for 4.1.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 2

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test]
```

Node 3 (offline node) Node 3 was online just long enough to be selected in the quorum of ballot #1 of Node 1.

B.2.6 Non-president goes offline before sending Voted message

The president retries the ballot here as well because it hasn't received all messages from the quorum in time (*Voted* in this case). After two attempts it updates the quorum, and the ballot succeeds.

Node 1

```
Received a new decree proposal for [Test]

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 1.1.
[Proposer] Received voted message from 3 for 1.1.

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 3.
[Proposer] Didn't get enough lastvote messages in time.
```

```
[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 2 (offline node)

```
> Test
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1.1] from 1
(exit)
```

Node 3

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test]
```

B.2.7 Non-president goes offline before receiving Success message

Ultimately it means that the offline node won't have the decree written in its ledger, which is exactly what the expected behavior is. Every quorum's node except for node 2 has the outcome (1:Test) written in their ledger.

Node 1

```
Received a new decree proposal for [Test]

[Proposer] Executing Paxos
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
```



```
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] Polling...
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 1.1.
[Proposer] Received voted message from 2 for 1.1.
[Proposer] Received voted message from 3 for 1.1.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 2 (offline node)

```
> Test
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
(exit)
```

Node 3

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test]
```

B.2.8 Multiple proposals at the same time

Execution of the steps is atomic, which means that once an action has begun, it has to be completed before another action can be started. This means that, for example, if multiple decrees are proposed at the same time, the first proposal's decree to arrive at the president's proposer will ultimately be written in the ledger. (That decree is Test3.)

Node 1

```
Received a new decree proposal for [Test3]
> Test1

Received a new decree proposal for [Test2]

[Proposer] Executing Paxos for decree=Test3
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
```

```
[Proposer] Polling...
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 1.1.
[Proposer] Received voted message from 2 for 1.1.
[Proposer] Received voted message from 3 for 1.1.
[Proposer] Ballot [1:Test3] succeeded.
[Learner] Written new decree [1:Test3]
[Proposer] Sending passed decree [1:Test3] to learners.
```

Node 2

```
> Test2
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test3]
```

Node 3

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
> Test3
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [1,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test3]
```

B.3 Multi-decree protocol test results

B.3.1 Proposing a series of decrees from different nodes

The main purpose of the multi-decree parliament is self-explanatory. We are testing a number of proposed decrees from different nodes. Node 1 is acting as president, and the other nodes are non-presidents taking part of the balloting. In the end, every proposed decree is written in the ledger.

Node 1

```
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 3.1
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 3.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.
```

```
[Proposer] No gaps to fill with olive day decrees.
> Test1

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=Test1, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Ballot [1:Test1] succeeded.
[Learner] Written new decree [1:Test1]
[Proposer] Sending passed decree [1:Test1] to learners.
Received a new decree proposal for [Test2]

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=2, decree=Test2, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Ballot [2:Test2] succeeded.
[Learner] Written new decree [2:Test2]
[Proposer] Sending passed decree [2:Test2] to learners.
Received a new decree proposal for [Test3]

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=3, decree=Test3, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Ballot [3:Test3] succeeded.
[Learner] Written new decree [3:Test3]
[Proposer] Sending passed decree [3:Test3] to learners.
Received a new decree proposal for [Test4]

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=4, decree=Test4, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Ballot [4:Test4] succeeded.
[Learner] Written new decree [4:Test4]
[Proposer] Sending passed decree [4:Test4] to learners.
> Test5

[Proposer] Executing Paxos
```

```

[Proposer] Polling...
[Proposer] Ballot: decreeId=5, decree=Test5, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
[Proposer] Ballot [5:Test5] succeeded.
[Learner] Written new decree [5:Test5]
[Proposer] Sending passed decree [5:Test5] to learners.

```

Node 2

```

[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test1]
> Test2
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [2:Test2]
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [3:Test3]
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [4:Test4]
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [5:Test5]

```

Node 3

```

[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test1]
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [2:Test2]
> Test3
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [3:Test3]
> Test4
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...

```

```
[Acceptor] Sending voted to 1
[Learner] Written new decree [4:Test4]
[Acceptor] Received beginballot [3,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [5:Test5]
```

B.3.2 Learning decrees - new president

Node 2 has decrees 1-3 written in its ledger. The president (Node 1) will send a *NextBallot*(b, n) message, where $n = 0$ since he has no decrees written in its ledger. In return, Node 2 will reply with a *LastVote* message including all decrees written in its ledger higher than $n = 0$, which is decrees 1-3. Node 1 will then proceed to conduct ballots for these decrees.

Node 1 (president)

```
=====Progress=====
LastTried = 3.1
PrevBal = -79228162514264337593543950335
PrevDec =
NextBal = -79228162514264337593543950335
=====Decreases=====
=====
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.

[Proposer] Learned about [1:Test1]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=Test1, quorum=2,1
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 4.1.
[Proposer] Received voted message from 2 for 4.1.
[Proposer] Ballot [1:Test1] succeeded.
[Learner] Written new decree [1:Test1]
[Proposer] Sending passed decree [1:Test1] to learners.

[Proposer] Learned about [2:Test2]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=2, decree=Test2, quorum=2,1
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 4.1.
```

```

[Proposer] Received voted message from 2 for 4.1.
[Proposer] Ballot [2:Test2] succeeded.
[Learner] Written new decree [2:Test2]
[Proposer] Sending passed decree [2:Test2] to learners.

[Proposer] Learned about [3:Test3]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=3, decree=Test3, quorum=2,1
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 4.1.
[Proposer] Received voted message from 2 for 4.1.
[Proposer] Ballot [3:Test3] succeeded.
[Learner] Written new decree [3:Test3]
[Proposer] Sending passed decree [3:Test3] to learners.

```

Node 2

```

=====Progress=====
LastTried = 3.2
PrevBal = 2.2
PrevDec = Test3
NextBal = 2.2
=====Decreases=====
Id=1, decree=Test1
Id=2, decree=Test2
Id=3, decree=Test3
=====

[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [4.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1

```

B.3.3 Learning decrees - non-president

After Node 3 receives a *NextBallot*(b, n) message from Node 1 (the president), Node 3 asks Node 1 for any decrees not in its ledger, lower or equal to n . Here, $n = 2$, and so therefore Node 1 will send the decrees 1 and 2 to Node 3.

Node 1 (president)

```

=====Progress=====
LastTried = 1.1

```

```

PrevBal = 1.1
PrevDec = Test2
NextBal = 1.1
=====Decreases=====
Id=1, decree=Test1
Id=2, decree=Test2
=====
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Sending nextBallotMsg to 3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Informing 3 with missing decrees: [1:Test1|2:Test2]
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.

```

Node 3 (non-president)

```

=====Progress=====
LastTried = -79228162514264337593543950335
PrevBal = -79228162514264337593543950335
PrevDec =
NextBal = -79228162514264337593543950335
=====Decreases=====
=====
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
Test1
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test1]
Test2
[Acceptor] Received beginballot [1.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [2:Test2]

```

B.3.4 Filling gaps

After the president learned decrees from other nodes, any missing decrees have to be filled by attempting to pass a ballot with an unimportant decree: the olive-day decree.

Node 1 (president)

```

=====Progress=====
LastTried = -79228162514264337593543950335
PrevBal = -79228162514264337593543950335
PrevDec =
NextBal = -79228162514264337593543950335

```

```

=====Decreases=====
=====
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Sending nextBallotMsg to 3,2,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.

[Proposer] Learned about [5:Test5]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=5, decree=Test5, quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [5:Test5] succeeded.
[Learner] Written new decree [5:Test5]
[Proposer] Sending passed decree [5:Test5] to learners.

[Proposer] Learned about [6:Test6]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=6, decree=Test6, quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [6:Test6] succeeded.
[Learner] Written new decree [6:Test6]
[Proposer] Sending passed decree [6:Test6] to learners.

[Proposer] Attempting to fill decree 1 with olive day decree.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=The ides of February is national olive day,
quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [1:The ides of February is national olive day] succeeded.
[Learner] Written new decree [1:The ides of February is national olive day]
[Proposer] Sending passed decree [1:The ides of February is national olive day]
to learners.

[Proposer] Attempting to fill decree 2 with olive day decree.

```



```

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=2, decree=The ides of February is national olive day,
quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [2:The ides of February is national olive day] succeeded.
[Learner] Written new decree [2:The ides of February is national olive day]
[Proposer] Sending passed decree [2:The ides of February is national olive day]
to learners.

[Proposer] Attempting to fill decree 3 with olive day decree.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=3, decree=The ides of February is national olive day,
quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [3:The ides of February is national olive day] succeeded.
[Learner] Written new decree [3:The ides of February is national olive day]
[Proposer] Sending passed decree [3:The ides of February is national olive day]
to learners.

[Proposer] Attempting to fill decree 4 with olive day decree.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=4, decree=The ides of February is national olive day,
quorum=3,2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 2.2.
[Proposer] Received voted message from 3 for 2.2.
[Proposer] Ballot [4:The ides of February is national olive day] succeeded.
[Learner] Written new decree [4:The ides of February is national olive day]
[Proposer] Sending passed decree [4:The ides of February is national olive day]
to learners.
[Proposer] Filled gaps in ledger with olive day decree.

```

Node 2

```

=====Progress=====
LastTried = -79228162514264337593543950335
PrevBal = -79228162514264337593543950335
PrevDec =
NextBal = -79228162514264337593543950335
=====Decreases=====
Id=5, decree=Test5
Id=6, decree=Test6
=====
[Server] Initialising...
[Server] Listening.

```

```

[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [1:The ides of February is national olive day]
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [2:The ides of February is national olive day]
[Acceptor] Received beginballot [2.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [3:The ides of February is national olive day]
[Learner] Written new decree [4:The ides of February is national olive day]

```

B.3.5 Overwriting olive decrees

In this scenario, Node 1 knows only of decree 4, and has olive-day decrees written for decrees 1-3. At the other end, Node 2 has all decrees 1-4 written. When Node 1 is elected, he learns about these decrees, begins balloting for these decrees, and writes these decrees in its ledger.

Node 1 (president)

```

=====Decrees=====
Id=1, decree=The ides of February is national olive day
Id=2, decree=The ides of February is national olive day
Id=3, decree=The ides of February is national olive day
Id=4, decree=Test4
=====

[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 1.1
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 3.1
[Proposer] Sending nextBallotMsg to 2,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.

```

```

[Proposer] Learned about [1:Test1]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=Test1, quorum=2,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Ballot [1:Test1] succeeded.
[Learner] Updated decree [1:Test1]
[Proposer] Sending passed decree [1:Test1] to learners.

[Proposer] Learned about [2:Test2]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=2, decree=Test2, quorum=2,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Ballot [2:Test2] succeeded.
[Learner] Updated decree [2:Test2]
[Proposer] Sending passed decree [2:Test2] to learners.

[Proposer] Learned about [3:Test3]. Conducting ballot.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=3, decree=Test3, quorum=2,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Ballot [3:Test3] succeeded.
[Learner] Updated decree [3:Test3]
[Proposer] Sending passed decree [3:Test3] to learners.

```

Node 2

```

=====Decrees=====
Id=1, decree=Test1
Id=2, decree=Test2
Id=3, decree=Test3
Id=4, decree=Test4
=====

[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 2.2
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Received beginballot [3.1] from 1

```

```
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
```

B.3.6 President shuts down after initiating ballot

Remember that steps 1-2 are done only once whenever a new president is assigned. With steps 1-2, the decree to propose is predetermined. Not in this case, since these steps are skipped for new decrees. Therefore, if a president shuts down before sending a *Success* message, the information is lost entirely.

Node 1

```
[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=test, quorum=2,3,1
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 3.1.
[Proposer] Received voted message from 2 for 3.1.
[Proposer] Received voted message from 3 for 3.1.
(exit)
```

Node 2

```
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Sending nextBallotMsg to 3,2,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
Received a new decree proposal for [Test]

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=1, decree=Test, quorum=3,2
[Acceptor] Received beginballot [4.2] from 2
[Acceptor] Voting for ballot...
```

```
[Acceptor] Sending voted to self
[Proposer] Received voted message from 2 for 4.2.
[Proposer] Received voted message from 3 for 4.2.
[Proposer] Ballot [1:Test] succeeded.
[Learner] Written new decree [1:Test]
[Proposer] Sending passed decree [1:Test] to learners.
```

Node 3

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Acceptor] Received new nextballot message.
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received beginballot [3.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
> Test
[Acceptor] Received beginballot [4.2] from 2
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 2
[Learner] Written new decree [1:Test]
```

B.3.7 Multiple proposals at the same time

In the multi-decree parliament, a sequence of proposed decrees should ultimately be written into the ledger. This test shows that for every proposed decree, a round of Paxos is executed.

Note: There was a performance issue related with high speed writing of separate decrees. Occasionally nothing would be done with a received *Success* message if many decrees were written one after another. This is likely because a new instance of the *Ledger* class has to be created, and for each *Ledger* instance, a save operation is done. This severely cripples performance.

A small adaptation to the Learner was made, which will make the Learner wait for a specified time-out, which is reset if it receives a new *Success* message. If (1) the time-out has been reached, and (2) there are decrees to be written, the decrees would be written all at once. Hence why many decrees are written all at once for Node 2 and 3.

Node 1

```
> Test1
Received a new decree proposal for [Test3]

[Proposer] Executing Paxos
[Proposer] Polling...
Received a new decree proposal for [test2]
[Proposer] Ballot: decreeId=1, decree=Test3, quorum=2,3,1
[Acceptor] Received beginballot [2.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
```

```

[Proposer] Received voted message from 1 for 2.1.
[Proposer] Received voted message from 2 for 2.1.
[Proposer] Received voted message from 3 for 2.1.
[Proposer] Ballot [1:Test3] succeeded.
[Learner] Written new decree [1:Test3]
[Proposer] Sending passed decree [1:Test3] to learners.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=2, decree=Test1, quorum=2,3,1
[Acceptor] Received beginballot [2.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 2.1.
[Proposer] Received voted message from 2 for 2.1.
[Proposer] Received voted message from 3 for 2.1.
[Proposer] Ballot [2:Test1] succeeded.
[Learner] Written new decree [2:Test1]
[Proposer] Sending passed decree [2:Test1] to learners.

[Proposer] Executing Paxos
[Proposer] Polling...
[Proposer] Ballot: decreeId=3, decree=test2, quorum=2,3,1
[Acceptor] Received beginballot [2.1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to self
[Proposer] Received voted message from 1 for 2.1.
[Proposer] Received voted message from 2 for 2.1.
[Proposer] Received voted message from 3 for 2.1.
[Proposer] Ballot [3:test2] succeeded.
[Learner] Written new decree [3:test2]
[Proposer] Sending passed decree [3:test2] to learners.

```

Node 2

```

> test2
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test3]
[Learner] Written new decree [2:Test1]
[Learner] Written new decree [3:test2]

```

Node 3

```

> Test3
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [2,1] from 1
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Acceptor] Received beginballot [2,1] from 1

```

```
[Acceptor] Voting for ballot...
[Acceptor] Sending voted to 1
[Learner] Written new decree [1:Test3]
[Learner] Written new decree [2:Test1]
[Learner] Written new decree [3:test2]
```

B.3.8 Taking leadership

A node becomes a president if it hasn't received a *Heartbeat* message within T minutes (Or 1 second in actual time. See Table 4.5.) with a **lower** id. Here we are testing the leadership election by turning nodes on and off in the following sequence:

1. Node 3 comes online, and becomes president
2. Node 2 comes online, and becomes president
3. Node 1 comes online, and becomes president
4. Node 1 goes offline, and node 2 becomes president
5. Node 2 goes offline, and node 3 becomes president

Node 1

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 3.1
[Proposer] Sending nextBallotMsg to 2,3,1,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 3.
[Proposer] Received lastvote from node 2.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
```

Node 2

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 2.2
[Proposer] Sending nextBallotMsg to 3,2,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
```

```
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 4,2
[Proposer] Sending nextBallotMsg to 3,2,
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 2.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
```

Node 3

```
[Server] Initialising...
[Server] Listening.
[Proposer] Preparing...
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 1,3
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 1)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Acceptor] Received new nextballot message.
[Acceptor] Sending LastVote to president (node 2)
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 5,3
```

B.3.9 Node joins the quorum after president received all LastVote messages

One of the major issues currently with my implementation of the *Multi-decree* protocol, is that the application comes to a halt if one of the non-presidents go offline. This is due to one of the requirements of the *StartPollingMajoritySet*: it requires all quorum members to have sent their *LastVote* message. So, if a new node joins in *after* the president executed steps 1-2, the balloting will fail and won't continue.

In this particular test case, Node 3 comes online, and Node 1 afterwards and becomes president. Node 1 prepares itself, and then Node 2 comes in and sends a proposal.

Node 1

```
[Server] Received leadership.
[Proposer] Learning decrees...
[Proposer] Ballot id incremented to: 1.1
[Proposer] Sending nextBallotMsg to 3,1,
[Acceptor] Received new nextballot message.
Set nextbal to 1.1
[Acceptor] Sending LastVote to self
[Proposer] Received lastvote from node 1.
[Proposer] Received lastvote from node 3.
[Proposer] Received all lastvotes from chosen set of nodes.
[Proposer] No gaps to fill with olive day decrees.
```



```
Received a new decree proposal for [Test]
```

```
[Proposer] Executing Paxos  
(Paxos stops here)
```

Node 2

```
[Server] Initialising...  
[Server] Listening.  
[Proposer] Preparing...  
> Test
```

Node 3

```
[Acceptor] Received new nextballot message.  
[Acceptor] Sending LastVote to president (node 1)  
[Acceptor] Requesting any potential missing decrees (id <= 0) from president.
```


Appendix C

Witness statements

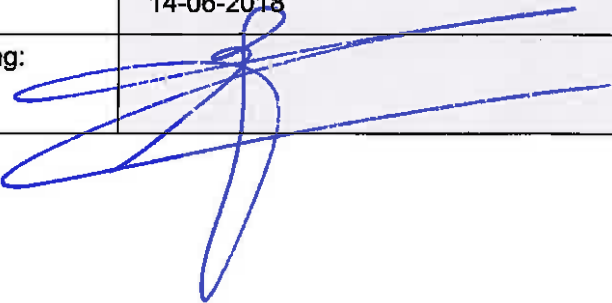
Getuigenverklaring bedrijfsbegeleider afstudeeropdracht

Naam student:	Florian van Herk
Studentnummer:	0898132
Opleiding:	Informatica
Titel eindverslag:	Paxos Blockchain – A private blockchain simulation based on the Paxos consensus algorithm
Getuigenverklaring bedrijfsbegeleider	
Waaruit blijkt dat de student professioneel en integer gedrag heeft vertoond?	
<i>De student heeft tijdens zijn werk professioneel en integer gedrag vertoond</i>	
Waaruit blijkt dat de student zich collegiaal heeft gedragen t.o.v. de medewerkers in uw organisatie?	
<i>De student heeft zich collegiaal gedragen en als vanzelfsprekend samengewerkt met andere studenten en medewerkers binnen het bedrijf</i>	
Waaruit blijkt dat de student zich aan organisatorische regels en planningen heeft gehouden?	
<i>De student heeft zich goed aan de gemaakte organisatorische afspraken gehouden en in overleg met zijn begeleider zijn planning en de bijstelling daarvan uitgevoerd</i>	
Waaruit blijkt dat de student voldoende werkoverleg heeft gehouden?	
<i>Daar waar nodig heeft er zowel op initiatief van de student als van zijn begeleider overleg plaatsgevonden</i>	
Op welke wijze heeft de student blijk gegeven van voldoende beheersing van kennis en vaardigheden om de opdracht uit te voeren?	
<i>In de met de student gevoerde gesprekken, zijn opgeleverde eindrapportage en de daarbij behorende pilot heeft de student blijk gegeven te beschikken over meer dan voldoende kennis en vaardigheden. De student heeft een nieuwe ontwikkeling i.c. blockchain technologie en het daarvoor benodigde consensus algoritmes vanuit de theorie geanalyseerd en omgezet naar werkende software</i>	
Op welke wijze heeft de student relevante actoren (klanten, ontwerpers, gebruikers en beheerders) betrokken bij het onderzoek en de resultaten?	
<i>De student heeft daar waar noodzakelijk overleg gevoerd met 'derden' en dit naar behoren gedaan</i>	
Hoe heeft de student kennis doorgegeven in uw organisatie?	
<i>De student heeft meerdere presentaties over zijn werk gegeven</i>	
Waaruit blijkt dat de student bruikbare adviezen heeft gegeven waarbij rekening gehouden is met de context en het perspectief?	
<i>De eindrapportage en de werking van het PAXOS consensus algoritme in de pilot heeft meer dan voldoende bijgedragen aan de kennis van het functioneren van dit algoritme binnen Centric</i>	
Op welke wijze heeft de student aan persoonlijke kennisontwikkeling gewerkt?	

De student heeft zich de theoretische kennis van het algoritme zelf eigen gemaakt en omgezet naar werkende software in NET

Bedrijf/organisatie:	Centric
Naam begeleider:	Prof. dr. Ben van Lier CMC
Datum:	14-06-2018
Handtekening:	

Getuigenverklaring opdrachtgever afstudeeropdracht

Naam student:	Florian van Herk
Studentnummer:	0898132
Opleiding:	Informatica
Titel eindverslag:	Paxos blockchain – A private blockchain simulation based on the Paxos consensus algorithm
Getuigenverklaring opdrachtgever	
Waaruit blijkt dat de student rekening heeft gehouden met functionele eisen en kwaliteitscriteria?	
<i>De student heeft zich goed gehouden aan de functionele eisen en de daarbij behorende criteria</i>	
Waaruit blijkt dat de student rekening heeft gehouden met de context van de dienst of het product?	
<i>De student heeft goed rekening gehouden met de context waarbinnen een consensus algoritme moet kunnen functioneren</i>	
Waaruit blijkt dat de student reviews/audits/functionele testen heeft laten uitvoeren of heeft uitgevoerd?	
<i>De software is getest en de tests en de daaruit voortvloeiende resultaten zijn beschreven in de eindrapportage</i>	
Waaruit blijkt dat de student met behulp van demonstraties, voorlichtingen, instructies of trainingen kennis heeft overgedragen?	
<i>De student heeft zowel binnen het SIA/KIEM traject van de HRO, bij Centric en in internationaal verband (Stuttgart) presentaties geven over zijn onderzoek en daaruit voortvloeiende software</i>	
Waaruit blijkt dat het eindresultaat (wel of niet) waardevol is?	
<i>Het resultaat van het onderzoek is buitengewoon waardevol en laat goed de werking zien van een consensus algoritme</i>	
Bedrijf/organisatie:	Centric
Naam begeleider:	Prof. dr. Ben van Lier CMC
Datum:	14-06-2018
Handtekening:	

Literature

- [1] Job Bakker. “Blockchain technology - An exploratory case study to identify the underlying principles and to determine the corresponding capabilities.” MA thesis. Leiden University, 2016.
- [2] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: (2008). URL: <https://bitcoin.org/bitcoin.pdf>.
- [3] Praveen Jayachandran. *Blockchain Explained - The difference between public and private blockchain*. 2017. URL: <https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/>.
- [4] Simone Vermeend, Perry Smit. *Blockchain: de technologie die de wereld radicaal verandert*. Einstein Books en Ebooks, 2017.
- [5] Leslie Lamport. “The Part-Time Parliament”. In: (1998). URL: <https://www.microsoft.com/en-us/research/publication/part-time-parliament/>.
- [6] Ben van Lier. *Blockchain, distributed ledgers and the Paxos protocol*. 2016. URL: <https://www.centric.eu/NL/Default/Themas/Blogs/2016/02/29/Blockchain-distributed-ledgers-and-the-Paxos-protocol->.
- [7] *DO-IT*. URL: <http://www.organic.nl>.
- [8] *Evo Fenedex - Over de vereniging*. URL: <https://www.evofenedex.nl/over-de-vereniging>.
- [9] *Van Reenen - Bedrijfsprofiel*. URL: <https://www.reenen.nl/over-ons/bedrijfsprofiel/>.
- [10] *Wie is Centric?* URL: <https://www.centric.eu/NL/Default/Over-Centric/Wie-is-Centric>.
- [11] *Centric Public Report 2017*. URL: <http://publications.centric.eu/centric-public-report-2017>.
- [12] *Pareltjes: Centric - Allrounder in het centrum van de Nederlandse ICT*. URL: <https://www.computable.nl/artikel/achtergrond/magazine/5627549/5215853/pareltjes-centric.html>.
- [13] Leslie Lamport, Robert Shostak, Marshall Pease. “The Byzantine Generals Problem”. In: (1982). URL: <https://lamport.azurewebsites.net/pubs/byz.pdf>.
- [14] Leslie Lamport, Eli Gafni. “Disk Paxos”. In: (2002). URL: <https://lamport.azurewebsites.net/pubs/disk-paxos.pdf>.
- [15] Leslie Lamport. “Fast Paxos”. In: *Distributed Computing* 19 (Oct. 2006). URL: <https://www.microsoft.com/en-us/research/publication/fast-paxos/>.
- [16] Leslie Lamport. “Paxos Made Simple”. In: (2001). URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [17] The Economist. “The great chain of being sure about things”. In: (2015). URL: <https://www.economist.com/news/briefing/21677228-technology-behind-bitcoin-lets-people-who-do-not-know-or-trust-each-other-build-dependable>.
- [18] *Leslie Lamport - A.M. Turing Award Laureate*. 2013. URL: https://amturing.acm.org/award_winners/lamport_1205376.cfm.

- [19] Leslie Lamport. *My writings*. 2017. URL: <https://lamport.azurewebsites.net/pubs/pubs.html>.
- [20] Leslie Lamport. “Time, Clocks and the Ordering of Events in a Distributed System”. In: (July 1978), pp. 558–565. URL: <https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/>.
- [21] *Most Cited Computer Science Articles*. URL: <http://citeseerx.ist.psu.edu/stats/articles>.
- [22] Leslie Lamport. *My writings*. 2017. URL: <https://lamport.azurewebsites.net/pubs/pubs.pdf>.
- [23] Mark Russinovich. *Framework for enterprise blockchain networks*. URL: <https://azure.microsoft.com/en-us/blog/announcing-microsoft-s-coco-framework-for-enterprise-blockchain-networks/>.
- [24] “The Coco Framework - Technical Overview”. In: (Aug. 2017). URL: <https://github.com/Azure/coco-framework/blob/master/docs/Coco%20Framework%20whitepaper.pdf>.
- [25] *What is the transport layer?* URL: <https://www.techopedia.com/definition/9760/transport-layer>.
- [26] Josh Stephens. *The Transport Layer: Understanding layer 4 of the OSI Model*. July 2011. URL: <https://www.computerworld.com/article/2470343/network-hardware-solutions/the-transport-layer---understanding-layer-4-of-the-osi-model.html>.
- [27] John Carbone , Express Logic. *Speed up machine-to-machine networking with UDP*. Dec. 2013. URL: <https://www.embedded.com/design/real-world-applications/4426378/Speed-up-machine-to-machine-networking-with-UDP>.
- [28] *Explanation of the Three-Way Handshake via TCP/IP*. Feb. 2010. URL: <https://support.microsoft.com/en-us/help/172983/explanation-of-the-three-way-handshake-via-tcp-ip>.
- [29] Nadeem Unuth. *TCP (Transmission Control Protocol Explained)*. Feb. 2018. URL: <https://www.lifewire.com/tcp-transmission-control-protocol-3426736>.
- [30] Bradley Mitchell. *TCP Headers and UDP Headers Explained*. Mar. 2018. URL: <https://www.lifewire.com/tcp-headers-and-udp-headers-explained-817970>.
- [31] Bradley Mitchell. *User Datagram Protocol - Understanding UDP and How It's Different From TCP*. Apr. 2018. URL: <https://www.lifewire.com/user-datagram-protocol-817976>.
- [32] T. Bova and T. Krivoruchka. “Reliable UDP Protocol”. In: (Feb. 1999). URL: <https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00>.
- [33] *UDPClient class*. URL: <https://docs.microsoft.com/nl-nl/dotnet/api/system.net.sockets.udpclient?view=netframework-4.7.1>.
- [34] JiaoLi Fang and Ming Liu. “Design and Implementation of Embedded RUDP”. In: *Design and Implementation of Embedded RUDP*. Kunming University of Science and Technology: IEEE, Sept. 2011. DOI: 10.1109/ICNDC.2011.9.
- [35] *SQLite - Write-Ahead Logging*. URL: <https://sqlite.org/wal.html>.