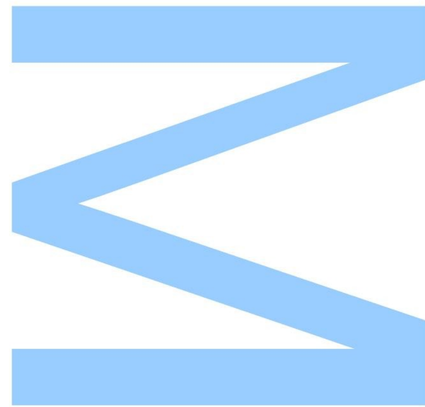
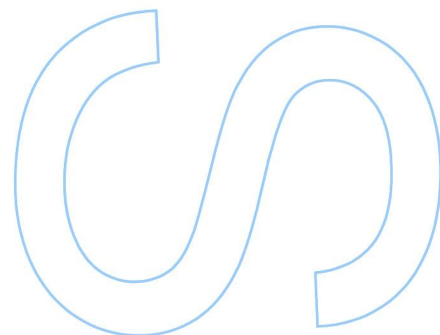
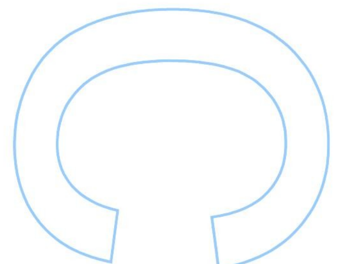
FORMAL VERIFICATION OF THE CEPH CONSENSUS ALGORITHM USING TLA+







Afonso das Neves Fernandes
Master's degree in Information Security
Computer Science Department
2021

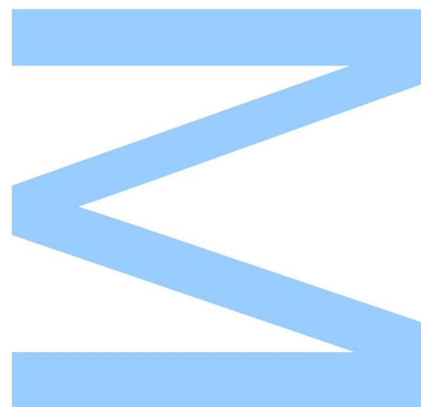
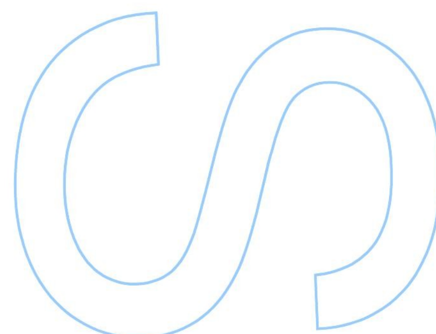
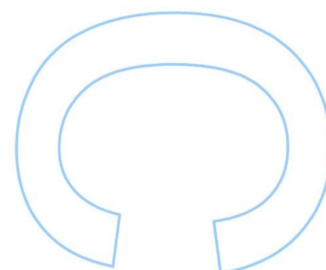
Supervisor
Rolando da Silva Martins, Assistant Professor,
Faculty of Sciences, University of Porto

Co-supervisor
Luís Filipe Coelho Antunes, Full Professor,
Faculty of Sciences, University of Porto

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Abstract

Nowadays, applications need to be designed to be scalable and resilient because the number of users is constantly growing. As a result of this, many applications are now designed as distributed systems, which grants scaling (using resources from multiple machines) and resilience (fault tolerance from using more than one machine). However, these systems come at the cost of being more error prone (source of error in the communication between the machines) and having longer and more complex implementations due to coordination challenges.

In this work, it is explored the use of formal methods as a pathway to help in the development and analysis of distributed applications. The system selected to explore the use of formal methods was Ceph, one of the most used open-source distributed storage platforms. Since the consensus algorithm is central to Ceph, it was formalized using the TLA$^+$ language. After this formalization, properties of the algorithm were verified using model checkers. The specification was also evaluated in terms of efficiency by counting the number of generated states.

The process of using formal methods was proven to be helpful as it was able to catch real world bugs in the consensus algorithm. To assess this, a bug that was present in an early version of Ceph was introduced in the specification. Then the specification was analyzed using a model checker to evaluate if the bug would be found, which it successfully managed to achieve. In this work, a visualization tool to explore a specification is also presented, which helps understand the algorithm, its bugs, and how a specification can be improved to make it more efficient.

Keywords: Distributed Systems, Consensus, Formal Methods.

Resumo

Atualmente, as aplicações precisam de ser concebidas para serem escaláveis e resistentes porque o número de utilizadores está em constante crescimento. Como resultado disto, muitas aplicações são agora concebidas como sistemas distribuídos, o que concede escalabilidade (utilizando recursos de várias máquinas) e resiliência (tolerância a falhas devido à utilização de mais do que uma máquina). No entanto, estes sistemas têm o custo de serem mais propensos a erros (fonte de erro na comunicação entre as máquinas) e terem implementações mais longas e mais complexas devido a desafios de coordenação.

Neste trabalho, é explorado o uso de métodos formais como um procedimento para ajudar no desenvolvimento e análise de aplicações distribuídas. O sistema seleccionado para explorar o uso de métodos formais foi o Ceph, uma das plataformas mais usadas de armazenamento distribuído e que é open-source. Uma vez que o algoritmo de consensus é central para o Ceph, o mesmo foi formalizado utilizando a linguagem TLA$^+$. Após esta formalização, as propriedades do algoritmo foram verificadas utilizando verificadores de modelos. A especificação foi também avaliada em termos de eficiência através da contagem do número de estados gerados.

O processo de utilização de métodos formais provou ser útil uma vez que foi capaz de avaliar e encontrar erros no algoritmo de consensus. Para avaliar isto, um erro que estava presente numa versão inicial do Ceph foi introduzida na especificação. Depois, a especificação foi analisada utilizando um verificador de modelos para avaliar se o erro seria encontrado, o que conseguiu alcançar com sucesso. Neste trabalho, uma ferramenta de visualização para explorar uma especificação também é apresentada, que ajuda a compreender o algoritmo, as suas falhas, e como pode ser melhorada uma especificação para a tornar mais eficiente.

Palavras-chave: Sistemas Distribuídos, Consensos, Métodos Formais.

Contents

Abstract i

Resumo ii

Contents v

List of Tables vi

List of Figures vii

Listings viii

Acronyms ix

1 Introduction 1

1.1 Motivation 1

1.2 Objectives 2

1.3 Contributions 2

1.4 Structure of the Document 3

2 Preliminaries 4

2.1 Distributed Systems 4

2.1.1 Architectural Organization 5

2.1.2 CAP Theorem 5

2.1.3 FLP Impossibility 6

2.1.4 Failure Models . . . 7
2.1.5 Summary . . . 7
2.2 Consensus in Distributed Systems . . . 7
2.2.1 Brief History . . . 8
2.2.2 The Problem . . . 8
2.2.3 Fault Thresholds . . . 9
2.2.4 Crash Fault Tolerant Algorithms . . . 11
2.2.5 Byzantine Fault Tolerant Algorithms . . . 14
2.2.6 Summary . . . 18
2.3 Formal Verification . . . 18
2.3.1 Specifying a System . . . 18
2.3.2 Model checking . . . 21
2.4 Related Work . . . 22

3 Case Study **23**
3.1 Introduction to Ceph . . . 23
3.1.1 Brief History . . . 23
3.1.2 Ceph Architecture . . . 23
3.1.3 Consensus in Ceph . . . 26
3.2 Ceph Consensus Algorithm Specification . . . 27
3.2.1 Design Choices and Specification Overview . . . 27
3.2.2 Variables and Constants . . . 28
3.2.3 Message Manipulation . . . 31
3.2.4 Transitions . . . 32
3.3 Verifying the Safety of the Algorithm . . . 44
3.3.1 Limiting the Space Exploration . . . 44
3.3.2 Invariants . . . 44
3.4 Algorithm Visualization . . . 45

3.4.1 TLA$^+$ Graph and Trace Explorer 45
3.4.2 Visualization of the Ceph Consensus Algorithm 48

4 Results and Analysis **50**
4.1 Testing the Specification 50
4.2 Performance of the Model 52
4.2.1 Results 52
4.2.2 Summary 55
4.3 Performance of the Visualization Tool 55

5 Conclusion **57**
5.1 Research Summary 57
5.2 Future Work 58
5.3 Conclusions 58

Bibliography **59**

A Specification Configuration **64**

List of Tables

2.1 Fault thresholds. 9

2.2 TLA$^+$ operators. 19

3.1 Constants in the specification. 29

3.2 Variables in the specification. 30

4.1 TLC execution statistics of the specification. 52

List of Figures

2.1 CAP theorem. 6
2.2 Paxos consensus message diagram. 12
2.3 Multi-Paxos consensus message diagram. 13
2.4 Byzantine Paxos consensus message diagram. 15
2.5 Practical Byzantine Fault Tolerance message diagram. 17

3.1 Ceph architecture. 24
3.2 Ceph data mapping. 25
3.3 Ceph monitor instance diagram. 26
3.4 Ceph consensus diagram. 26
3.5 State graph example. 46
3.6 Trace error file example. 46
3.7 Usage example of the TLA$^+$ Graph Explorer. 49

4.1 Number of states depending on the maximum epoch. 53
4.2 Number of states depending on the maximum proposal number. 53
4.3 Number of states depending on the maximum number of committed values. . . . 54
4.4 Number of states depending on the number of monitors in the system. 54
4.5 Memory used depending on the size of the file. 56

Listings

2.1 Formatting conjunctions and disjunctions in TLA$^+$. 19
2.2 TLA$^+$ specification of an insecure transaction. 20
2.3 TLA$^+$ configuration file for the specification in the Listing 2.2. 21
2.4 TLC output for the specification in the Listing 2.2. 21
3.1 Type aliases in the specification. 29
3.2 Message manipulation operators. 31
3.3 Next statement. 32
3.4 Message dispatcher. 33
3.5 Leader election. 34
3.6 Definition of the operators for a monitor crash and recover. 34
3.7 Operators used to start a collect phase. 35
3.8 Operator that generates proposal numbers. 36
3.9 Operator to handle a collect message. 36
3.10 Excerpt of the operator to handle a last message. 37
3.11 Operator that ends collect phase. 38
3.12 Operators that handles a client request. 39
3.13 Operators to handle a client request. 40
3.14 Excerpt of the operator to handle a begin message. 40
3.15 Excerpt of the operator to handle an accept message. 41
3.16 Excerpt of the operator to send the commit messages. 41
3.17 Operator used to commit values shared in a commit message. 42
3.18 Operators used to start a lease phase. 43
3.19 Operator to handle a lease message. 43
3.20 Operator to limit the search space. 44
3.21 Safety invariants. 44
3.22 HTML template to render the information of a monitor. 48
4.1 Modifications to the specification to introduce the bug. 50

Acronyms

BFT Byzantine Fault-Tolerance

CRUSH Controlled Replication Under Scalable Hashing

MDS Metadata Server

OSD Object Storage Device

PBFT Practical Byzantine Fault-Tolerance

TLA Temporal Logic of Actions

RADOS Reliable Autonomous Distributed Object Storage

RTOS Real-Time Operating System

Chapter 1

Introduction

1.1 Motivation

As the world digitalizes, applications must serve an increasing number of people and requests, which normally comes at a cost of computing power. Besides this, applications are also being developed and used in critical scenarios that have demanding requirements, for example, that require low downtime (which can be achieved using fault tolerance).

To respond to the previous requirements, many applications are now being developed using distributed systems. However, with this paradigm comes a set of challenges. These systems tend to be more error prone (more sources of errors) which can make the coordination between the machines more challenging.

To reach agreement and coordination between the machines in a distributed system, many algorithms have been devised. However, these algorithms are complex and hard to implement. Formal verification is a process which can help with these tasks by providing a way to design and verify a complex system and ensure its correctness and resilience.

In software systems, formal verification is defined as the process of specifying a system in a formal language and afterwards verifying its properties. A formal specification of a system has the advantages of providing a clearer representation of the system, where all the actions are discriminated, and providing the ability to use tools that automatically check properties of the system.

This technique is used throughout the industry for designing critical system. One of the languages that stands out in the formal verification paradigm is TLA$^+$, which has been used to design components of Amazon Web Services [32] and Microsoft Azure [31]. TLA$^+$ is used in the designing step of a system, sometimes described as the step between the implementation and describing the system in prose, and helps find errors at the design level that are normally more complex and that would be harder to find by directly testing the implementation, as described in the works that was previously mentioned.

1.2 Objectives

The main objective of this work is to formalize and verify the consensus algorithm used in Ceph [41]. Ceph is one of the most used open-source distributed storage platforms, for example is used by CERN [37] to store the petabytes of information created by the large hadron collider.

One of the main advantages of Ceph, compared to other storage solutions, is its resilience over data loss in the presence of failure of some machines of the Ceph cluster. To accomplish this, Ceph uses a consensus algorithm that is based on Paxos [21], although with some modifications. This was the main reason to choose this system since, even if Paxos has proofs of correctness, it is also important to verify if the changes made to the algorithm do not compromise its correctness.

Furthermore, with this work we also aim to:

- Help understand how the Ceph's core consensus algorithm works.
- Verify properties of the consensus algorithm and check its correctness.
- Make it easier to evaluate new versions/iterations of the consensus algorithm.

To help in the objective of understanding how the algorithm works, a visualization tool was also developed, which can be used for any formalized specification.

Initially, one of the goals was to propose a Byzantine Fault-Tolerance (BFT) protocol to replace the protocol for consensus in Ceph. The initial research phase was done, however it was decided to focus on the specification and verification of the current protocol before designing and prototyping a new one.

1.3 Contributions

This work contributes to the community with two open-sourced projects hosted on GitHub[1]. The first project has the formal specification of the consensus algorithm used in Ceph, while the second project has the visualization tool that was developed, which can be used for exploring a formal specification.

The work done in this thesis was also presented in the 2021 TLA$^+$ Conference in St. Louis, Missouri[2], the main conference for the TLA$^+$ language, with the presence of both the maintainers and the creator of the language, Leslie Lamport.

[1] https://github.com/afonsonf/ceph-consensus-spec and https://github.com/afonsonf/tlaplus-graph-explorer
[2] http://conf.tlapl.us/

1.4 Structure of the Document

This document is organized in the following chapters:

- Chapter 2 presents the background material required to read this work: theory of distributed systems; algorithms for distributed consensus; introduction to formal verification; and related work.
- Chapter 3 describes the work done in this thesis: presentation of Ceph and the algorithm used for consensus; specification of the formalized algorithm; verification of properties of the algorithm; and presentation of the visualization tool.
- Chapter 4 presents the results and analysis on the performance of the specification and the visualization tool.
- Chapter 5 lays down the main conclusions and some ideas for future work.

Chapter 2

Preliminaries

2.1 Distributed Systems

A distributed system can be defined in several ways, however, in this work, the definition used is the one given in Tanenbaum and Steen book [36]:

> A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

This definition points to two important aspects of a distributed system. First, the existence of multiple computing elements (commonly referred to as nodes) that can be applications on the same or different machines. Second, the collaboration between the nodes have to appear to its users as a coherent system.

When designing a distributed system, the benefits and drawbacks of using such paradigm must be considered, as well as what can be obtained from it.

Some goals when designing a distributed system should be:

- **Resource sharing**, the ability of the system to use distributed resources to improve its performance or reliability.
- **Transparency of distribution**, simplifying the use of distributed resources to the user of the system.
- **Scalability**, the possibility to scale up or down the system as needed.

There are also some challenges that should be accounted, that arise when designing type of system, such as: reliability, bandwidth, and latency of the network; heterogeneity of the system; and administrative costs.

In this section, some important definitions and results about distributed systems are discussed.

2.1.1 Architectural Organization

Architectural organization is a characteristic that can be used to classify distributed systems. Three types of organizations are considered in this work: Centralized, Decentralized, and Hybrid organizations.

In a **centralized organization**, a system can be divided into servers and clients. In this architecture, the client and servers are separated entities. This is the one more typically used because of its simplicity and because it covers the requirements of most systems.

In a **decentralized organization**, there is no division into clients and servers (nodes in the system have both roles). These systems are commonly referred to as peer-to-peer systems. They are used when a centralized component is not wanted, for example, because of security or resilience issues.

Finally, a **hybrid organization** is when there is a peer-to-peer system with centralized components. These components can have different purposes, from coordination servers to primary content distribution servers. Hybrid systems can be used, for example, in mobile edge scenarios, as described in Iris [30].

2.1.2 CAP Theorem

An important theorem when working with distributed systems that have shared storage or a common state is the CAP theorem. Gilbert and Lynch [13] proved this theorem, where they state that a distributed system, as described before, can only guarantee two of the three following properties at the same time:

- **Consistency** (C): The ability for all the nodes to be at the same state at all the time.
- **Availability** (A): The ability for the system to be always available to receive requests.
- **Partition Tolerance** (P): The ability to be tolerant to partitions in the network.

The reasoning is that a system can not be both available and consistent in the presence of partitions. If availability is chosen, there is no guarantee that that the modifications made to a partition are replicated to the other partitions, losing consistency. If consistency is chosen, modifications can not be allowed while the system is partitioned, losing availability. The compromises of not having one of the three properties are shown in Figure 2.1.

In practical scenarios, systems must choose between consistency and availability (or make some assumptions in one of them) because network failures can always occur and cause partitions. The algorithms explored in this work, generally choose consistency over availability [22], that is, in presence of failures of some nodes, the system may become temporally unavailable while it recovers (chooses a new leader or makes some reconfiguration).

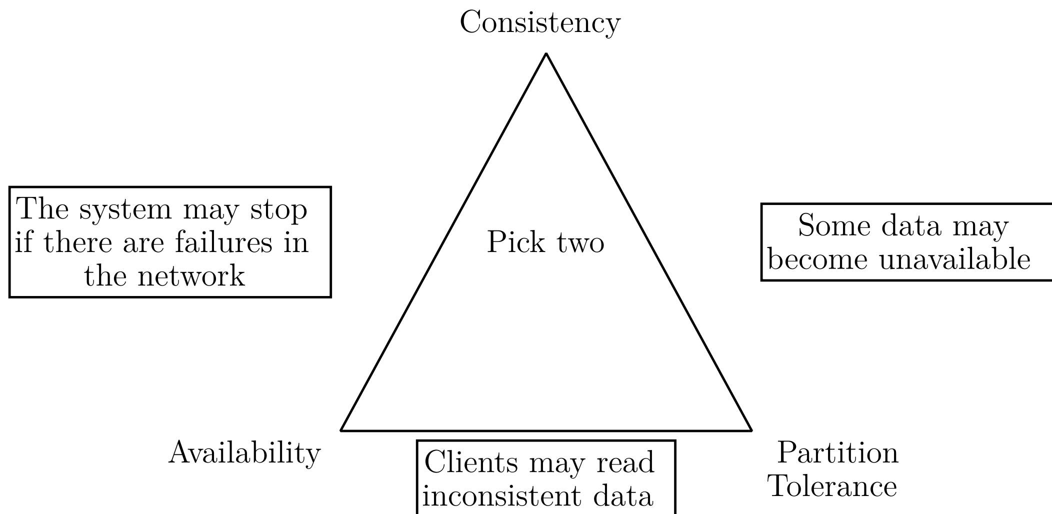


Figure 2.1: Visualization of the CAP theorem.

2.1.3 FLP Impossibility

The FLP Impossibility was presented by Fischer, Lynch, and Paterson [12]. It states that in an asynchronous system of nodes, where one or more nodes may fail, any algorithm for them to agree on a value may not finish.

There are two important concepts presented in the previous definition: the asynchronous system and the fault type. An asynchronous system is a system where there are no bounds for the time it takes to process or deliver a message by a node. The fault type considered are unannounced fail-stops, where a node fails by silently stopping the communication.

The reasoning is that a node can not differentiate between a slow node and a failed one. This can lead to situations where a node will wait endlessly for another one and cause the protocol/algorithm to not finish.

In a practical scenario, some assumptions and modifications about the system model can be made in order to be possible to achieve consensus. The most common modification is to create timeouts, if a node takes more than a certain amount of time to respond consider it failed. The possibility of non-termination can also be minimized by improving the network or using randomized algorithms [2].

After the FLP Impossibility was published, several research was done to try answer the minimum synchrony required for consensus [10]. Dwork, Lynch, and Stockmeyer [11] defined the partial synchrony required for fault-tolerant consensus. They showed that consensus can be achieved in a partially synchronous system by separating the safety and liveness properties of the protocol. This is done by ensuring that in the asynchronous periods of the protocol safety is not violated (with the loss of some liveness properties).

2.1.4 Failure Models

A distributed system is an enabling environment for various kinds of failures. To better prevent failures, it is important to know the main types of failures and what are the main characteristics of each type.

A failure model explanation will be given based on what was described by Hadzilacos and Toueg [14]. Three types of failures are considered in this work: Crash, Omission, and Byzantine failures. However, there will be more emphasis, in this work, on Crash failures.

In a **Crash Failure**, the node halts the computation and stops responding.

In an **Omission Failure**, the node fails to respond to a request. This happens, for example, when a node does a computation and does not process the result or receives some information and does not handle it.

A **Byzantine Failure** is any behaviour that a node has that differs from the expected/normal behaviour. This type covers the failures described before and failures that result from malicious nodes.

2.1.5 Summary

As stated before, distributed systems are used when resource sharing is necessary, for example scaling the system or making it more resilient. Nowadays, many systems are designed to be distributed, mainly using replication to support failures or peaks of traffic.

The focus of this work will be distributed storage systems, particularly the challenge of distributed consensus, which is characteristic in these systems. In the next section, the challenge of distributed consensus and the current solutions for this problem are explained.

2.2 Consensus in Distributed Systems

In distributed systems, consensus is the problem of multiple nodes agreeing on a sequence of values [8]. Solutions to this problem, consensus algorithms, can be used to create distributed systems where all correct nodes execute the same commands, in the same order, reaching the same state.

The design of a consensus algorithm is particularly challenging when considering the possible failures of the nodes. A distributed database system is an example where consensus is crucial to order the transactions (at each step, choosing the transaction that will be committed). If a faulty network is considered, the consensus algorithm will have to decide what to do with out of order messages.

2.2.1 Brief History

In the early eighty's, many research was made in the area of distributed consensus and several protocols were proposed [18, 19]. The first protocol to stand out was paxos, devised in 1989 and later published in 1998 [21]. This protocol is still widely used in the industry [1], and is similar to the Viewstamped Replication protocol, devised around the same time for database replication [33].

The Practical Byzantine Fault-Tolerance (PBFT) protocol was published in 1999 [7]. This protocol started a great interest in BFT protocols [16, 40]. Nowadays, with the increasing scale of systems (such as the appearance of big networks of computers in blockchains), more research has been done in this area and there were more protocols published. An example of recent work in the area is Raft [34], a crash fault tolerance protocol designed to be simpler and easier to understand than paxos.

2.2.2 The Problem

As stated before, distributed consensus is the problem of multiple nodes agreeing on a sequence of values. This problem can be reduced to a simpler problem of multiple nodes agreeing on a single value, normally referred as single-value consensus [8]. The general consensus can be obtained by executing a sequence of instances of the single-value consensus.

Some important properties of single-value consensus are [22]:

- The chosen value must be a value from the set of proposed values.
- Only a single value must be chosen.
- A correct node will only learn that a value was chosen if it actually has been chosen.

The consensus problem is generally solved by electing a leader or coordinator node, for better efficiency. This node then tries to choose a value by proposing it to the other nodes, and if they accept it, it chooses the value and propagates that information to the system.

Challenges arise when considering failures in the system. For example, in a crash fault model, how the system recovers when a leader fails or how it resolves multiple leaders proposing different values, in the presence of partitions. In a byzantine fault model, there are more challenges to consider, for example, how the system tolerates and recovers from a compromised leader or a node that says one thing and does another.

One of the first papers to state this problem was published by Lamport [27]. The problem stated in the paper is the "Byzantine Generals Problem" and it is because of this paper that byzantine failures are called byzantine.

In the paper, the authors describe a situation in which three generals try to decide whether

to attack the enemy camp. The attack is only successful if the three of them attack. They show that, if one of the generals is compromised (has a byzantine failure), consensus can not be reached. The next section shows how many nodes are required for consensus and the corresponding proofs.

2.2.3 Fault Thresholds

A deciding characteristic when comparing protocols is the number of nodes required to tolerate f failures. There are studies that provide us with the minimum of nodes required for each type of system model and supported failure [11, 27]. In Table 2.1, there is a description of the minimum number of nodes required, in a partially synchronous system, for different type of failures, adapted from [5].

Table 2.1: Fault thresholds for different system models, addapted from [5].

Failure type	Partially synchronous
Crash	$2f + 1$
Byzantine	$3f + 1$

Note that there are some Byzantine Fault-Tolerance (BFT) algorithms that use only $2f + 1$ nodes. These algorithms consider a system with a hybrid failure model where, in some parts of the algorithm, it is assumed that there are no byzantine failures by using, for example, trusted components [40].

To understand the proof of the thresholds, it is required to first define the properties of liveness and safety of a system.

A system guarantees **safety** if it ensures that wrong events never take place. This ensures that all correct replicas execute the same sequence of operations [5].

A system guarantees **liveness** if it ensures that good events will eventually take place [39]. Namely, a correct client request will eventually be executed.

The following proofs will be adapted from CS244b: Distributed Systems course notes at Stanford University[1]. A quorum is a subset of nodes that participate in a decision.

Lemma 2.2.1. *The minimum of number nodes required to tolerate f nodes with crash failures is* $2f + 1$.

Proof. Consider a system with N nodes, f of which may suffer crash failures, and quorums of size Q.

To guarantee liveness, there must be a non-failed quorum. Thus, $Q \leq N - f$, the size of the quorum must be, at most, the number of non failed nodes.

[1] http://www.scs.stanford.edu/14au-cs244b/notes/pbft.txt

To guarantee safety, any two quorums must intersect in at least one node. The system will diverge if two quorums do not intersect. Hence, $2Q - N > 0$.

By joining the previous two restrictions, we have: $N < 2Q \leq 2N + 2f$. Simplifying, we have $N > 2f$, and the smallest number N can take is $2f + 1$. With $N = 2f + 1$, the smallest size for a quorum is $f + 1$.

□

Example: Consider a system with three nodes, where one of the nodes may suffer a crash failure. Let's say nodes A, B, and C, where A may fail.

If a quorum size of one node is considered, then safety will not be guaranteed. Node A can, alone in a quorum, respond to a request and then fail, losing the information of the request.

With a quorum size of three nodes, there is no guarantee of liveness. Node A can fail and there will be no quorum available.

With a quorum of two nodes, both properties can be guaranteed. Consider nodes A and B make a quorum and respond to a request. After the request, even if A fails, node C can learn about the request from node B, guaranteeing safety. Liveness is guaranteed because it is assumed that with three nodes only one of them may fail, that is, B and C can always make a quorum.

Lemma 2.2.2. *The minimum of number nodes required to tolerate f nodes with byzantine failures is* $3f + 1$.

Proof. Consider a system with N nodes, f of which may suffer byzantine failures, and quorums of size Q.

To guarantee liveness there must be a quorum with correct nodes. Thus, $Q \leq N - f$.

To guarantee safety, any two quorums must intersect in a **correct** node. Hence, $2Q - N > f$.

Joining both restrictions, we have: $N + f < 2Q \leq 2(N - f)$. Simplifying the inequation, we have: $N + f < 2N - 2f \Leftrightarrow N > 3f$. The smallest number for the size of the system is $3f + 1$, and with this size, the smallest quorum size will be $2f + 1$.

□

Example: Consider a system with four nodes, where one of the nodes may suffer a byzantine failure. Let's say nodes A, B, C and D, where A may fail.

The explanation for why quorum sizes of one and four do not work is identical to the scenario in the previous proof.

With a quorum size of two nodes, safety will not be guaranteed. Let's assume nodes A and B make quorum to respond to a request. Consider also that node A lies about the result of the

request (Byzantine failure). Then, nodes C and D will not know in which node to trust, losing the request.

With a quorum size of three nodes, both properties are guaranteed. Let's assume nodes A, B, and C make quorum to respond to a request. Even if A lies about the result, D can know the true result by majority (nodes B and C will have the same result).

2.2.4 Crash Fault Tolerant Algorithms

A Crash Fault Tolerant Algorithm can run "flawlessly" in a system where some crash failures may occur. This means that, even if some failures occur, the algorithm will run while maintaining some safety and liveness properties.

In this section, the paxos algorithm is explained, one of the first and most used crash fault tolerant algorithms for consensus.

2.2.4.1 Paxos

The paxos algorithm was designed by Lamport and published in 1998 [21]. Lamport first explains an algorithm to solve the single-value consensus, which will be referred to as the paxos consensus algorithm. Then, using the paxos consensus algorithm, Lamport designs an algorithm to solve general consensus, which will be referred to as the multi-paxos or paxos algorithm.

The paxos algorithm works in a system with:

- Crash fault model, where nodes work at arbitrary speed, may fail by stopping and may restart.
- Asynchronous communication model, where messages can take arbitrary long to be delivered, can be duplicated and can be lost, but not corrupted. For liveness, some periods of synchrony are required.

The algorithm requires $2f + 1$ nodes to tolerate f failures and that any two quorums intersect in at least one node. In the algorithm a node may have one or more of the following roles: proposer, acceptor, and learner.

A proposer is responsible for leading voting processes. The proposer suggests a value to the acceptors and receive their votes. If enough acceptors vote favorably, the proposer can give instructions to commit the value. This node is normally chosen in an election.

An acceptor is responsible for voting in proposals. The role of an acceptor is to resolve conflicts (and reach consensus) when there is more than one proposal being sent. An acceptor also shares his previous committed and uncommitted values to guarantee that information is passed between rounds (from crashes).

A learner is responsible for executing a proposal that has been voted by enough acceptors (generally referred to as committing a value).

The paxos consensus algorithm is divided into four phases: 1a, 1b, 2a, and 2b (represented in Figure 2.2). Before the algorithm, and whenever a node suspects the proposer has failed, an election process occurs. The election process is leaved as an implementation detail in [22], and the author mentions the election can be done by using either randomness or real time (timeouts).

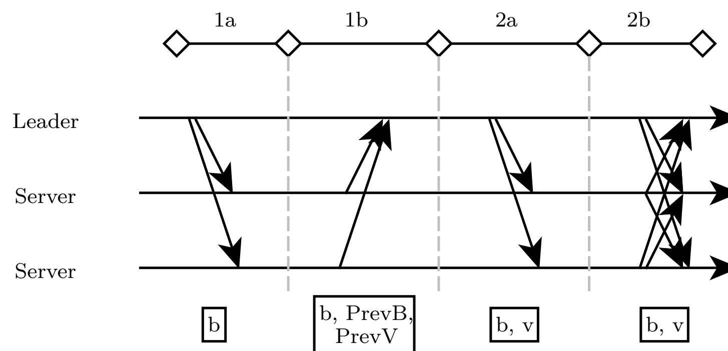


Figure 2.2: Paxos consensus message diagram. Each server is both an acceptor and a learner. One of the servers, the Leader, is also a proposer. In the diagram, b is a ballot number, v a value, $PrevB$ a previous voted ballot number and $prevV$ a previous voted value.

The algorithm proceeds as follows:

- Phase 1a: The proposer chooses a ballot number b, which will identify the proposal, and sends it in a 1a message to a majority of acceptors.
- Phase 1b: The acceptor will accept the 1a message with a ballot number b if b is greater than all previously ballot numbers that he accepted. If the message is accepted, the acceptor sends an 1b message with the biggest ballot it has committed, and the value corresponded to that ballot.
- Phase 2a: When the proposer receives 1b messages from a quorum of acceptors he chooses a safe value to commit in ballot number b. If no value has been previously committed, he can choose any value, else he will choose the value with the greatest ballot number that was previously committed. The proposer sends a message to the acceptors with the ballot number and the chosen value.
- Phase 2b: The acceptor will accept the 2a message with a ballot number b if b is greater or equal than all previously ballot numbers that he accepted. If the message is accepted, the acceptor broadcasts a 2b message with the ballot number and value.
- Commit: A value v is chosen (or committed) when there is a quorum of acceptors that send 2b messages with the value v.

The algorithm presented above solves the single-value consensus problem, that is, there will be only one value chosen (or committed). If there are quorums of acceptors that sent 2b messages for ballot numbers $b1$ and $b2$, then the values chosen in those ballots are the same.

The key to the why the algorithm works is in phase 2a, when the proposer chooses a value to propose. The following properties on what value is safe to propose at ballot b ensure that only one value will ever be chosen [25]:

- If no acceptor in a quorum has previously voted in a ballot numbered less than b, then all values are safe to propose.
- If some acceptor in a quorum has previously voted, then the value in the highest ballot he voted is safe to propose.

The solution to general consensus, multi-paxos, is obtained by executing a sequence of instances of the paxos consensus algorithm. This can be done by dividing the decisions in slots and attaching to the messages an identification of the slot for which the value will be chosen.

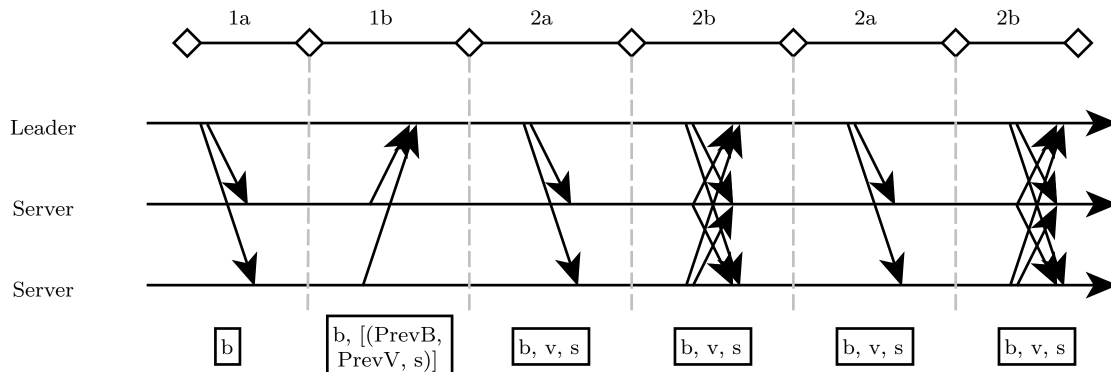


Figure 2.3: Multi-Paxos consensus message diagram. Each server is both an acceptor and a learner. One of the servers, the Leader, is also a proposer. In the diagram, b is a ballot number, v a value, s a slot, $PrevB$ a previous voted ballot number and $prevV$ a previous voted value.

The modifications to the single-value consensus algorithm are the following:

- Phase 1b: The acceptor sends, for each slot, the biggest ballot it has committed, and the value corresponded to that ballot.
- Phase 2a: The proposer first chooses a slot and then a safe value to commit in that slot. If no value has been previously committed in the ballot number b and slot s, he can choose any value, else he will choose the value with the greatest ballot number that was previously committed in the slot s.

- Phase 2b: The 2b messages are sent also with corresponded slot number.
- Commit: The commit is executed for a pair value and slot.

The first phase only needs to be executed once for each proposer. After this phase, the proposer can start, with the ballot number chosen in the first phase, multiple second phases with different slot numbers, as presented in Figure 2.3.

The safety property in the general consensus algorithm is that only one value will be chosen per slot. That is, if there are quorums of acceptors that sent 2b messages for ballot numbers $b1$ and $b2$ with the same slot number, then the values chosen in those ballots are the same.

Optimizations

There are some optimizations that can be made to the algorithm, although with some compromises. Some examples are Cheap Paxos [26] and Fast Paxos [24], both presented by Leslie Lamport.

In Cheap Paxos, f of the $2f + 1$ nodes are considered auxiliary and only take part in the protocol to reconfigure the system (used by the main nodes to distinguish between failed nodes and failed communication medium). This optimization comes at the expense of liveness. For example, consider a system of two main nodes p and q, and a third auxiliary node (tolerance to one node failure). If p fails, and then q fails while p is recovering, the system will halt until p is repaired [26].

In Fast Paxos, there is one less communication step, the client sends the request directly to the acceptors instead of the client sending it to the proposer and then the proposer to the acceptors. This optimization comes at the expense of needing f more nodes, that is $3f + 1$ nodes to tolerate f failures. The quorum needs to be bigger because, in the case of concurrent client requests, the acceptors may receive them in different order and end in a situation similar to the one explained in the byzantine threshold proof (Section 2.2.3).

2.2.5 Byzantine Fault Tolerant Algorithms

A Byzantine Fault Tolerant Algorithm can run in a system where byzantine failures may occur while maintaining liveness and safety properties.

In this section, two BFT algorithms for consensus are explained. A byzantine version of paxos explained by Lamport in [25], that is easier to follow from the previous presented algorithm (Paxos) and PBFT, presented by Castro and Oki in [7], that is vastly implemented in the industry [35].

2.2.5.1 Byzantine Paxos

The Byzantine Paxos was presented by Lamport in [25], where Lamport explains the process of changing an algorithm that tolerates crash failures to also tolerate byzantine failures (Byzantizing Paxos). Using this refinement, Lamport shows how the PBFT can be obtained from paxos.

The algorithm that is explained in this section corresponds to the BPCon in the paper, and it solves the single-value consensus problem. The algorithm to solve the general consensus problem can be obtained using the same idea presented in previous section, running multiple instances of the algorithm that are identified by different slots.

Byzantine Paxos assumes a system with the same asynchrony model as the paxos algorithm and with a byzantine fault model, where nodes may fail by stopping, send arbitrary messages and may restart.

The algorithm requires $3f+1$ nodes to tolerate f failures and that any two quorums intersect in at least $f+1$ nodes. Each node can have the same roles presented in the paxos algorithm: proposer, acceptor, and learner. The algorithm is divided in six phases: 1a, 1b, 1c, 2a, 2av, and 2b (represented in Figure 2.4).

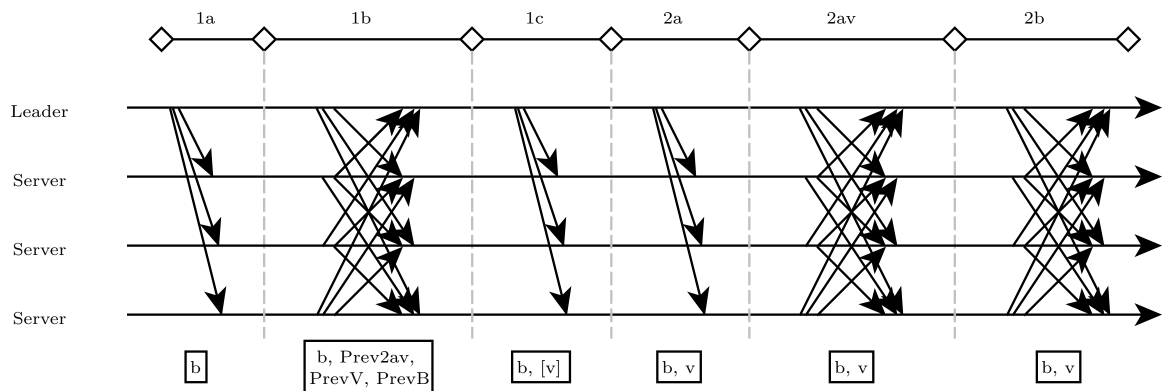


Figure 2.4: Byzantine Paxos consensus message diagram. Each server is both an acceptor and a learner. One of the servers, the Leader, is also a proposer. In the diagram, b is a ballot number, v a value, $PrevB$ a previous voted ballot number, $prevV$ a previous voted value, and Prev2av the set of previous 2av messages that the server has sent.

The algorithm proceeds as follows:

- Phase 1a: The proposer chooses a ballot number b, which will identify the proposal, and sends it in a 1a message to a majority of acceptors.
- Phase 1b: The acceptor will accept the 1a message with a ballot number b if b is greater than all previously ballot numbers that he accepted. If the message is accepted, the acceptor

broadcasts a 1b message with the biggest ballot it has committed, the value corresponded to that ballot, and a set with the 2av messages that he previously sent.

- Phase 1c: When the proposer receives 1b messages from a quorum of acceptors, he broadcasts 1c messages with the values that are safe to commit. An acceptor will reject 1c messages with unsafe values.

  If there is an acceptor that voted for a value v in a ballot c less than b and, (a) did not vote for any ballot greater than c and, (b) there are $f+1$ 1b messages from acceptors with a 2av message saying that they voted for v in a ballot greater or equal than c, then v is safe to commit. Else, if no acceptor in a quorum has previously voted in a ballot numbered less than b, any value is safe to commit.

- Phase 2a: The proposer chooses one value v from the values that he sent in a 1c message. Then he sends it in a 2a message with ballot number b to a majority of acceptors.
- Phase 2av: The acceptor will accept the 2a message with value v and a ballot number b if b is greater or equal than all previously ballot numbers that he accepted and if v is in a 1c message that he has received and not rejected. If the message is accepted, the acceptor broadcasts a 2av message with the ballot number and value.
- Phase 2b: If an acceptor received 2av messages from a quorum of acceptors with value v and ballot number b, he broadcasts 2b message with v and b.
- Commit: A value v is chosen (or committed) when there is a quorum of acceptors that send 2b messages with the value v.

In the algorithm presented above, there will be only one value chosen (or committed), that is, if there are quorums of acceptors that sent 2b messages for ballot numbers b1 and b2, then the values chosen in those ballots are the same. Therefore, solving the single value consensus problem.

The key properties that ensure that there will be only one value chosen, similar to paxos, are the verifications an acceptor does when deciding if a value is safe or not to commit in a ballot b [25]:

- If no acceptor in a quorum has previously voted in a ballot numbered less than b, then all values are safe to commit.
- If there is an acceptor that voted for a value v in a ballot c less than b and, (a) did not vote for any ballot greater than c and, (b) there are $f+1$ 1b messages from acceptors with a 2av message saying that they voted for v in a ballot greater or equal than c, then v is safe to commit.

The version of this algorithm that solves the general consensus problem is equivalent to the PBFT algorithm that is explained in the next section.

2.2.5.2 Practical Byzantine Fault Tolerance

The Practical Byzantine Fault Tolerance algorithm was designed by Castro and Oki in 1999 [7]. It is an algorithm that solves the general consensus problem in a system with a byzantine fault model. One of the main differences from Byzantine Paxos is that this algorithm is presented with more implementation details, however, in this section, the algorithm is explained without those details to make it easier to understand.

The algorithm works in a system with the same asynchrony model and fault model as the Byzantine Paxos. It requires $3f + 1$ nodes to tolerate f byzantine failures and that any two quorums intersect in at least $f + 1$ nodes.

Each node can have one of two roles: Primary or Backup. In each view (corresponds to the ballot) there is one primary and remaining nodes are backups. The primary is assigned in a round robin way, the primary of view v is $v \ mod \ |R|$, where R is the set of nodes.

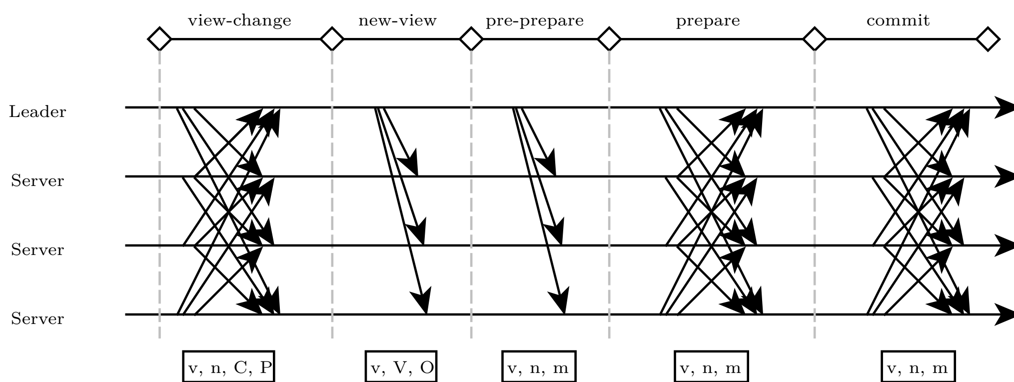


Figure 2.5: Practical Byzantine Fault Tolerance message diagram. One of the servers, the Leader, is the primary and the reaming servers are backups. In the diagram, v is a view number, m a value, n a sequence number, C a set of previous checkpoint messages, P a set of previous prepare messages, V a set of view-change messages, and O a set of pre-prepare messages.

The algorithm works in a similar way to the Byzantine Paxos algorithm. The corresponding phases between the two are:

- Phase 1a: There is no explicit 1a phase, the new view is triggered cooperatively by timeouts.
- Phase 1b: Equivalent to the view-change phase, to synchronize with the other nodes.
- Phase 1c: Corresponds to the new-view that ends the view transition phase.
- Phase 2a: Corresponds to the pre-prepare phase, to propose a value to be committed.
- Phase 2av: Equivalent to the prepare phase, to vote on the value to commit.
- Phase 2b: Equivalent to the commit phase to announce that a node has committed a value.

Like in multi-paxos, the first two phases, the view transition, only need to be executed once per view. After the view transition, the algorithm works as follows [7]: The client sends a request to the primary; The primary broadcasts the requests to the backups (pre-prepare and prepare); The replicas execute the request and send reply to the client (commit); The client waits for $f+1$ replies with the same result.

2.2.6 Summary

Consensus is an important problem that needs to be addressed when building distributed systems. In this sections some solutions were presented, however, these solutions can be complex and hard to implement and test. In the next section, formal verification methods are presented as a way to test the properties of consensus algorithms and the corresponding implementations.

2.3 Formal Verification

Formal verification, in the context of software systems, is the act of reasoning about a system or algorithm to prove its correctness, a set of properties it should guarantee [23]. This can be divided into two steps: first writing a specification that describes the algorithm and then checking the specification, either by state enumeration (model checkers) or by writing proofs checked by automatic systems.

There are several languages (accompanied by the corresponding tools) to do formal verification, such as: Dafny [28], Coq [4], and TLA$^+$ [20]. The language chosen to be used in this work was TLA$^+$, an extended version of Temporal Logic of Actions (TLA) presented in [20]. This was the language that was decided on because of the extensive use in the industry, including on verifying systems that are similar to the one that will be verified in this work.

In this section, the steps to do formal verification are explained, using TLA$^+$. First, the overall process of writing a specification is explained, and then how that specification can be checked using model checking. There is also a proof system to writing and checking proofs in TLA$^+$, however that was not explored in this work.

2.3.1 Specifying a System

A system is described in TLA$^+$ as a state machine, that is, a system is specified as a set of variables that have one or more initial states and change in discrete steps. The syntax is based on first order logic and uses operators such as conjunctions, disjunctions, and quantifiers.

There is also a higher level syntax to write specifications that is similar to C code, named PlusCal. Those specifications can then be translated to TLA$^+$ to be executed. However, PlusCal will not be explained because all the specification were written directly using TLA$^+$.

Syntax and types

In this section, the main types and operators in TLA$^+$ that are used in the specifications in this work are explained. For a complete list of types and operators supported read [23].

The main types that a variable in TLA$^+$ can have are:

- Base types like booleans, numbers, and strings.
- Sets (e.g. {}, {1,2,3}): A collection of unique items with the same type.
- Sequences (e.g. «1,2,"a"»): A ordered list of items.
- Functions (e.g. 3 :> 1, [x \in 1..3 |-> 0]): A mapping of a set of items with the same type in a set of items that can have different types.
- Records (e.g. [type |-> 0, value |-> "A"]): A structure to hold multiple values.

The main operators in TLA$^+$ are listed in Table 2.2.

Table 2.2: TLA$^+$ operators.

Operator	Example	Description
/\	A /\ B	Conjunction, true if A and B are true.
\/	A \/ B	Disjunction, true if A or B are true.
=>	A=> B	Implication.
@@	(2 :> 1) @@ [x \in 1..3 |-> 0]	Function constructor, can be used to create new functions (similar to merging or adding elements).
=	A = B	Equality, true if A equals B.
#	A # B	Difference, true if A is not equal to B.
<, <=, >, >=	A < B	Comparisons.
==	Foo(A,B) == A+B	Definition, defines Foo(A,B) to be A+B.
\E	\E x \in S: P(x)	Existence, true if exists x in S such that P(x) is true.
\A	\A x \in S: P(x)	True if P is true for all elements in S.

Some important notes on the syntax:

- A variable with a prime (e.g. var') corresponds to the value of the variable in the next state.
- In a transition, the value of all variables in the next state needs to be defined. UNCHANGED var is the same as var' = var.
- Multiple conjunctions or disjunctions can be formatted in a vertical way. For example, in Listing 2.1, `Expression1` is equivalent to `Expression2`.

```
1 Expression1 == A /\ (B \/ C)
2 Expression2 == /\ A
3               /\ \/ B
4                  \/ C
```

Listing 2.1: Formatting conjunctions and disjunctions in TLA$^+$.

Example

In Listing 2.2 is a specification of an algorithm to make transactions. The transactions can be submitted to a list of servers. Each user starts with 10 units of money.

In lines [2-7] the variables and constants are defined, and their initial value. In lines [9-32] the main actions of the system are defined. And in the line 34 a safety invariant is defined that states that every user must have a positive balance.

The version of the algorithm in the specification is insecure because it violates the safety invariant. In the next section this specification is checked with a model checker to find an example where the safety invariant is violated.

```
1  ------------------------------ MODULE transaction ------------------------------
2  CONSTANTS Users, Servers
3  VARIABLES User_Balance, Server_State, Pending_Transaction
4
5  Init == /\ User_Balance = [u \in Users |-> 10]
6          /\ Server_State = [s \in Servers |-> "IDLE"]
7          /\ Pending_Transaction = [s \in Servers |-> 0]
8
9  SubmitTransaction(From, Dest, Amount, Server) ==
10     /\ From # Dest
11     /\ Server_State[Server] = "IDLE"
12     /\ User_Balance[From] >= Amount
13     /\ Server_State' = [Server_State EXCEPT ![Server] = "Transaction"]
14     /\ Pending_Transaction' = [Pending_Transaction EXCEPT ![Server] =
15         [From |-> From, Dest |-> Dest, Amount |-> Amount]]
16     /\ UNCHANGED User_Balance
17
18 ExecuteTransaction(Server) ==
19     /\ Server_State[Server] = "Transaction"
20     /\ LET From   == Pending_Transaction[Server].From
21            Dest   == Pending_Transaction[Server].Dest
22            Amount == Pending_Transaction[Server].Amount
23        IN  User_Balance' = [u \in Users |->
24                                 CASE u = From -> User_Balance[u] - Amount
25                                   [] u = Dest -> User_Balance[u] + Amount
26                                   [] OTHER    -> User_Balance[u] ]
27     /\ Server_State' = [Server_State EXCEPT ![Server] = "IDLE"]
28     /\ Pending_Transaction' = [Pending_Transaction EXCEPT ![Server] = 0]
29
30 Next == \/ \E server \in Servers: \E u1, u2 \in Users: \E amount \in 1..20:
31             SubmitTransaction(u1, u2, amount, server)
32         \/ \E server \in Servers: ExecuteTransaction(server)
33
34 Inv == \A user \in Users: User_Balance[user] >= 0
35 ================================================================================
```

Listing 2.2: TLA$^+$ specification of an insecure transaction.

2.3.2 Model checking

Model checking is the act of verifying if a model of a specification has a set of properties. In TLA$^+$ the model checker program is named TLC [43]. TLC is the model checker used in this work and it does the verification by enumeration of reachable states. There is also a new model checker for TLA$^+$ in early development, Apalache [17], that verifies a specification using constraints solvers, such as Microsoft Z3 [9].

```
1 CONSTANTS Users = {user1, user2}
2           Servers = {srv1, srv2}
3 INIT Init
4 NEXT Next
5 INVARIANT Inv
```

Listing 2.3: TLA$^+$ configuration file for the specification in the Listing 2.2.

A TLA$^+$ specification is usually accompanied by a configuration file that defines values to be used in the model checker. In Listing 2.3 is the configuration file for the specification in the previous section. It defines the constants Users and Servers to be sets of two elements each. It also defines the predicate for the initial state, the next state transitions, and the invariant.

```
1 (...)
2 Error: Invariant Inv is violated.
3 Error: The behavior up to this point is:
4 (...)
5 State 3: <SubmitTransaction line 13, col 5 to line 19, col 29 of module
    transaction>
6 /\ Server_State = (srv1 :> "Transaction" @@ srv2 :> "Transaction")
7 /\ Pending_Transaction = ( srv1 :> [From |-> user2, Dest |-> user1, Amount |->
    1] @@ srv2 :> [From |-> user2, Dest |-> user1, Amount |-> 10] )
8 /\ User_Balance = (user1 :> 10 @@ user2 :> 10)
9
10 State 4: <ExecuteTransaction line 22, col 5 to line 31, col 72 of module
    transaction>
11 /\ Server_State = (srv1 :> "IDLE" @@ srv2 :> "Transaction")
12 /\ Pending_Transaction = (srv1 :> 0 @@ srv2 :> [From |-> user2, Dest |-> user1,
    Amount |-> 10])
13 /\ User_Balance = (user1 :> 11 @@ user2 :> 9)
14
15 State 5: <ExecuteTransaction line 22, col 5 to line 31, col 72 of module
    transaction>
16 /\ Server_State = (srv1 :> "IDLE" @@ srv2 :> "IDLE")
17 /\ Pending_Transaction = (srv1 :> 0 @@ srv2 :> 0)
18 /\ User_Balance = (user1 :> 21 @@ user2 :> -1)
19
20 2880 states generated, 1853 distinct states found, 1372 states left on queue.
21 The depth of the complete state graph search is 5.
```

Listing 2.4: TLC output for the specification in the Listing 2.2.

In Listing 2.4 is depicted a summary of the output that TLC prints for the previous configuration and specification. TLC reports an error stating that the invariant in the specification is violated and gives an example of a behaviour that leads to the violation. In this example, the error comes from two transactions being executed in concurrency without a lock system.

2.4 Related Work

Formal verification is highly adopted in the designing of new complex systems and algorithms. In this section, some works of using TLA$^+$ in the industry are mentioned, namely at Amazon, Intel, Microsoft, and in the design of consensus algorithms such as Raft and Tendermint.

The usage of TLA$^+$ at Amazon Web Services was reported in a paper [32], where they explain how they use formal methods and why they chose TLA$^+$. Here, they report the usage of TLA$^+$ in multiple services and that it helped them find various design bugs. It also helped them by giving confidence in the correctness of changes in the algorithms to make aggressive optimizations without sacrificing correctness.

At Intel, the usage of TLA$^+$ was reported in the paper [3], written by an Intel engineer and the TLA$^+$ author. In this work they explain the uses of TLA$^+$ in the industry, namely in designing multiprocessor protocols both at Intel and at Digital and Compaq.

Microsoft has used TLA$^+$ in several Azure products, one of them is the Cosmos DB, a globally distributed database [31]. In the specification of this product, Microsoft defines five consistency levels, with trade-offs in availability and performance that are incorporated in different service options, of the Cosmos DB, tailored for different business requirements.

The distributed Real-Time Operating System (RTOS), OpenComRTOS, the successor of Virtuoso that was used by the European Space Agency's Rosetta spacecraft, was designed using TLA$^+$ [38]. In this work they explain how they used formal methods and present the various improvements obtained compared to the previous version of the distributed RTOS.

TLA$^+$ has also been used to ensure and prove correctness of recent complex algorithms and protocols. The Raft algorithm, a crash fault tolerant algorithm, was presented with a TLA$^+$ specification to prove its correctness in [34]. The Tendermint protocol, a BFT protocol, was verified using TLA$^+$ in [6]. The Pastry algorithm for a distributed hash table was verified using TLA$^+$ [29]. The Paxos algorithm has also been specified in TLA$^+$ by Lamport [25], however this was for the general abstract algorithm and in this work the focus will be on the implemented version in Ceph.

Chapter 3

Case Study

3.1 Introduction to Ceph

The amount of data produced and collected is growing, and with it, the storage requirements for the systems are also increasing [15]. One of the most well-known solutions for distributed storage, and that is widely adopted in the industry, is Ceph [41]. In particular, is is used internally by the department of computer science of Faculty of Science, University of Porto as its main storage backend. This section serves as an introduction to Ceph. First, a brief history of Ceph development and use is presented. Afterwords, the architecture and main components of Ceph are described. Finally, the role of consensus in Ceph is explained.

3.1.1 Brief History

Ceph was created by Sage Weil and presented in his doctoral dissertation [41], in 2007. After his graduation, he continued to work on Ceph creating Inktank Storage in 2012 to deliver professional services and support for Ceph. Inktank was later bought by Red Hat in 2014, which has since then been one of the major maintainers and contributors to Ceph. Ceph is an open-source project and is backed by several major companies like Red Hat, Canonical, and Intel.

Some of the major points that make Ceph stand out from the competition are being open source, scalable to petabytes of storage, and the ability to run in commodity hardware. Because of this, Ceph has been adopted by many companies, and one of the major users is CERN [37], where they use it to store the petabytes of information created by the large hadron collider.

3.1.2 Ceph Architecture

Ceph is a solution for distributed storage that is composed of multiple components. An overview of the architecture of Ceph is illustrated in Figure 3.1.

In the client side, Ceph is composed of a library named Librados that can be used to communicate directly with the various Object Storage Device (OSD) within the cluster. In the server side, there are multiple components that are part of the Reliable Autonomous Distributed Object Storage (RADOS) cluster.

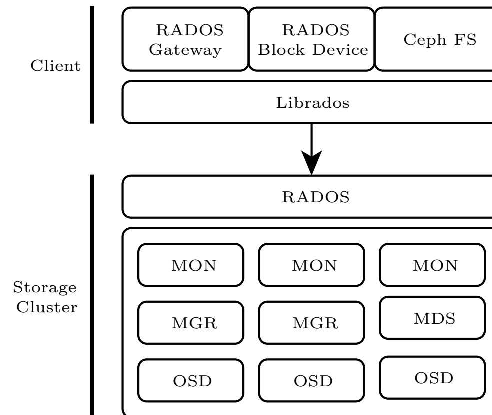


Figure 3.1: Diagram of the Ceph architecture. MON stands for monitor servers, MGR stands for manager servers, MDS stands for metadata servers, and OSD stands for object storage devices.

It is possible to interact with a storage cluster using multiple interfaces. The storage interfaces supported, using the Librados library, are:

- RADOS Gateway, an object storage interface that provides applications with a RESTful gateway to the storage cluster. The interface API is compatible with Amazon S3 service and OpenStack Swift.
- RADOS Block Device, an interface that provides the ability to provision traditional block devices that can be used in applications such as virtual machines or Kubernetes. The block devices can be stored over multiple OSDs.
- Ceph File System, an interface that provides a POSIX-compliant file system interface to use file systems on top of RADOS. This interface uses the MDS servers to access the file metadata.

A Ceph storage cluster has the following components:

- Monitors, responsible for maintaining maps of the state of the cluster. This includes the monitor map, manager map, OSD map, MDS map, and the CRUSH map. To maintain these maps, monitors use a version of the Paxos algorithm. For redundancy and high availability, it is recommended to have between three and seven monitors.
- Managers, responsible for keeping track of the cluster metrics such as system load and storage utilization. The manager provides a web-based dashboard and a REST API. For high availability there should be at least two managers.

- Object Storage Device, responsible for storing the data and handling data replication, recovery, and rebalancing. For high availability and redundancy, at least three OSDs are required.
- Metadata Server (MDS), responsible for storing file metadata. This server is only required if the interface Ceph FS is used. The MDS allows the user to run commands like ls and find with less stress on the storage cluster. A cluster can have multiple MDSs for high availability and better performance.

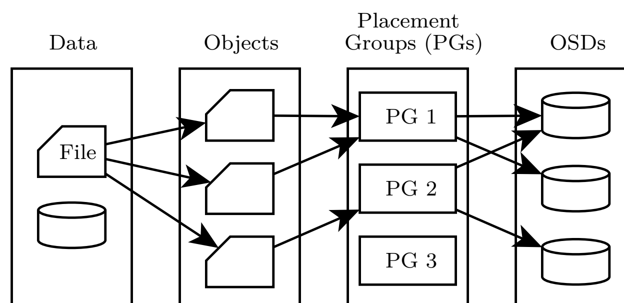


Figure 3.2: Diagram explaining how data is stored in a pool of a storage cluster.

The action of storing data in a pool of a storage cluster (either files, block storage, or object storage) follows the process presented in Figure 3.2.

In the first step, the data is divided into smaller objects. Then, each object is added to a pool and assigned to one of the available placement groups. Finally, the placement groups are assigned to OSDs.

To assign the placement groups to OSDs, Ceph uses the Controlled Replication Under Scalable Hashing (CRUSH) algorithm [42]. This is a deterministic algorithm that distributes the placement groups while taking into account some arguments.

Example of arguments that the CRUSH algorithm considers are:

- Failure domain, defines the level of failure tolerance of the storage cluster (examples are osd, host or rack). For example, if the failure domain is host, then the replicas of a PG will be assigned to different hosts.
- Number of replicas, defines the number of failures the system can tolerate.
- Replication strategy, defines if the strategy is either replication or using erasure code technique.
- Type of storage, defines the type of OSDs the pool should use, such as HDD or SSD.

This is the procedure to add data to one Ceph pool, and there can be multiple Ceph pools, each one with its configuration (such as number of placement groups and rules for the CRUSH algorithm).

3.1.3 Consensus in Ceph

In a Ceph storage cluster, the monitors are responsible to maintain the maps of various resources. These maps contain information about the resources available in the cluster and their status. This information is crucial to Ceph because CRUSH uses it to compute where each data object is in the cluster, and if this information is invalid, clients will not know where their data is.

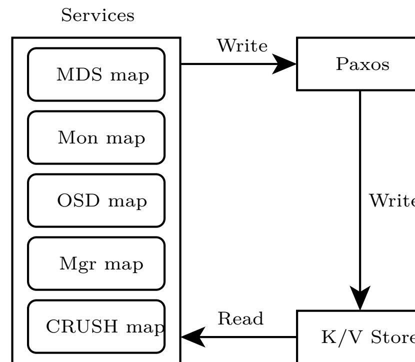


Figure 3.3: Diagram of a Ceph monitor (based on a image from the Ceph blog[1]).

For this motive, the monitors use a version of the Multi-Paxos algorithm [21] to maintain maps of the OSDs, MDSs, monitors, and managers in the cluster. Ceph uses the Multi-Paxos algorithm in order to maintain consistent versions of these maps in the presence of failures. In Figure 3.3 is a representation of the services inside a monitor instance. Each of those services can make requests to the paxos instance, which interacts with other monitors to reach consensus.

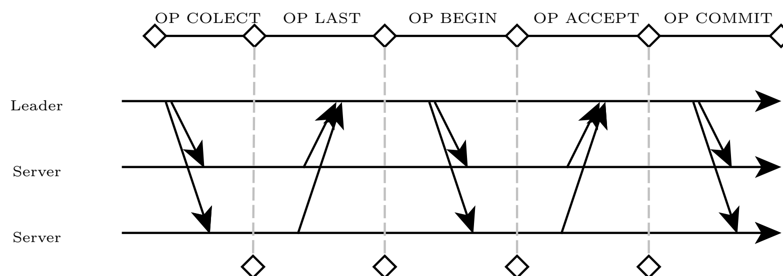


Figure 3.4: Diagram of the Ceph consensus algorithm.

In Figure 3.4 is a diagram of the consensus algorithm used in Ceph. Like Multi-Paxos, the algorithm is divided in two phases. The first part is a recovery phase (`OP COLLECT` and `LAST`), and this phase occurs only once, after the leader is elected. After a successful recovering phase, the leader can start consequent committing phases (`OP BEGIN`, `ACCEPT`, and `COMMIT`) to commit values. This is a simplified version, the complete one is explained in the next section.

[1] https://ceph.com/dev-notes/cephs-new-monitor-changes/

3.2 Ceph Consensus Algorithm Specification

Formal specification is a powerful tool to analyse complex algorithms and check their correctness. In this section, the TLA$^+$ formal specification of the Ceph consensus algorithm is described. First, an overview of the specification and some design choices are presented and then each section of the specification is explained. The complete specification is hosted on GitHub[2].

3.2.1 Design Choices and Specification Overview

In the context of this work, there were two main goals to write the specification of the Ceph consensus algorithm. First, to have a formal specification of the implemented algorithm, which can be used to study and better understand how the algorithm works (and in the future study improvements). Second to verify the correctness of the algorithm, check if it holds a set of properties in every possible state of the system.

For these motives, the specification was written and designed to model, in the best possible way, the implemented algorithm, and to abstract only implementation details. The environment where the algorithm runs was also taken into consideration (for example considerations about the message protocol used between the replicas and what properties it ensured). An overview of these design choices is explained below.

Communication. In Ceph, monitors use TCP to communicate between each other. This ensures that there is no message duplication and there are no messages delivered out of order (for a given connection). Therefore, in the specification, the messages were modelled using a set of connections (one for each pair of monitors), where each connection holds a queue of messages to be delivered. A message can be lost if a monitor crashes or if a timeout occurs.

Election of a leader and quorums. The leader is chosen from the active monitors in a given election (the specification models all possible leader choices). There is only one leader elected per term (term is also referred to as epoch). In the specification, the current epoch is stored in a global variable that is shared between monitors. Similar to what happens in the implementation, odd epochs correspond to an election and even epochs correspond to the consensus algorithm execution. Therefore, new elections can be triggered by incrementing the epoch variable. In an election, the quorum that will form will have all available (not crashed) monitors.

Transactions (values committed). In the Ceph consensus algorithm, the values that are proposed and committed are called transactions. These transactions are composed of a set of changes to values in the stable store. To simplify this, in the specification, the values committed are abstracted as a single value change, from a set of possible values that can be proposed. This is, an abstracted client can request a value, from a set of values, to the leader and the leader will execute the algorithm to try to change the value in his store to the one requested.

[2]https://github.com/afonsonf/ceph-consensus-spec

Failure model. The Ceph consensus algorithm is tolerant to crash failures of f nodes for a cluster of $2f+1$ nodes. This is, the algorithm will ensure safety properties even if some nodes fail by stopping. There were two ways considered to model this in the specification. First to make it explicit by having variables that track if a node is down. The other way was to let it be checked by the model checker when searching for reachable system states. The option chosen was the first one because the second one would deviate from the algorithm implemented and would make it harder to visualize the algorithm. In the future, the specification can be refined to the second option, which may lead to better performance.

The specification works as follows. The system starts in an initial configuration where all variables have default values, and the system is in an election (epoch equal to 1). From there, the specification will follow consequent next actions/transitions. If the system is in an election period it will elect a leader and increment the epoch, else it will make an action of the algorithm or call a timeout or monitor crash/recover.

After the election, the system enters a recovery period where the monitors will take actions to agree on a proposal number (collect and last messages). When the monitors agree on a proposal number, they will take one of two actions. If there is a value that was being committed, and the commit was stopped (for example by a timeout), the leader will propose it and start a commit phase (if it is a valid value). Else the monitors enter an active state (like an idle state) waiting for new requests.

The commit phase starts by the leader sharing the value to be committed and asking the monitors in the quorum if they agree on the value (messages begin and accept). If the leader receives positive responses from the monitors, he will commit the value and send messages to the monitors to instruct them to also commit it (message commit).

Between these actions, a monitor may crash or recover, or a timeout can occur, triggering new elections. These events are also formalized as actions of the specification.

The specification is divided into sections. The first section has declaration of variables and constants. Then a section with helper predicates, including the ones for message manipulation. Then the main functions of the algorithm, divided in sections for each phase. Followed by a section with the predicates for leader election and one for timeout and monitor crashes/recover. Finally, a section with dispatchers (handlers for each message type), and the next statement that includes all the possible transitions.

3.2.2 Variables and Constants

The specification starts by declaring all the constants and variables that are used. Constants are for values that are constant, and the value of a constant is defined in the configuration file. Variables have values that can change between transitions. The initial value of a variable is defined in an Init predicate, and between transitions all values of the variables in the next state must be explicitly defined.

Table 3.1: Constants in the specification.

Constant	Type	Description
Monitors	Set(MONITOR)	Set of monitors in the cluster.
Value_set	Set(VALUE)	Set of possible values to commit.
Nil	VALUE	Reserved null value.
STATE_RECOVERING, STATE_ACTIVE, STATE_UPDATING, STATE_UPDATING_PREVIOUS, STATE_WRITING, STATE_WRITING_PREVIOUS, STATE_REFRESH, STATE_SHUTDOWN	STATE_NAME	Monitor states (as they appear in the implementation).
PHASE_ELECTION, PHASE_SEND_COLLECT, PHASE_COLLECT, PHASE_LEASE, PHASE_LEASE_DONE, PHASE_BEGIN, PHASE_COMMIT	PHASE_NAME	Monitor phases. They are used in the specification (not in the implementation) to force some sequence of steps.
OP_COLLECT, OP_LAST, OP_BEGIN, OP_ACCEPT, OP_COMMIT, OP_LEASE, OP_LEASE_ACK	MESSAGE_OP	Messages types.

In Tables 3.1 and 3.2 is a summary of the constants and variables used in the specification, respectively. The types are written using the Snowcat language from the Apalache type checker[3]. The type annotations can be used to check type consistency using the tool previously mentioned.

Some types are abstracted, such as with MONITOR and VALUE, where the value can be a string, an integer or something else. Other types have type aliases where the type is defined in further annotations, such as with PN and MESSAGE.

In Listing 3.1 are the type aliases used in the specification, including the extended type of a message.

```
1  @typeAlias: VALUE_VERSION = Int;
2
3  @typeAlias: PN = Int;
4
5  @typeAlias:  MESSAGE =
6    [type: MESSAGE_OP, from: MONITOR, dest: MONITOR,
7     first_committed: VALUE_VERSION, last_committed: VALUE_VERSION,
8     values: (VALUE_VERSION -> VALUE), uncommitted_pn: PN, pn: PN];
9
10 @typeAlias: MESSAGE_QUEUE = MONITOR -> (MONITOR -> Seq(MESSAGE));
```

Listing 3.1: Type aliases in the specification.

[3] https://apalache.informal.systems/docs/apalache/typechecker-snowcat.html

Table 3.2: Variables in the specification.

Constant	Type	Description
epoch	Int	Integer representing the current epoch. If is odd trigger an election.
messages	MESSAGE_QUEUE	Messages waiting to be handled.
message_history	Set(MESSAGE)	History of messages exchanged.
quorum	MONITOR -> Bool	Stores if a monitor is up or down. All available monitors, in a given epoch, are part of the quorum.
quorum_sz	Int	Size of the current quorum.
isLeader	MONITOR -> Bool	If a monitor is the current leader.
state	MONITOR -> STATE_NAME	The state of each monitor.
phase	MONITOR -> PHASE_NAME	The phase of each monitor.
pending_pn	MONITOR -> PN	Stores a proposal number when the commit phase starts.
pending_v	MONITOR -> VALUE_VERSION	Stores a value version when the commit phase starts.
uncommitted_pn	MONITOR -> PN	Stores the best uncommitted pn received in the collect phase.
uncommitted_v	MONITOR -> VALUE_VERSION	Stores the best uncommitted value version received in the collect phase.
uncommitted_value	MONITOR -> VALUE	Stores the best uncommitted value received in the collect phase.
monitor_store	MONITOR -> VALUE	The value committed in the storage of each monitor.
values	MONITOR -> (VALUE_VERSION -> VALUE)	The transaction log of each monitor.
accepted_pn	MONITOR -> PN	The last proposal number accepted by each monitor.
first_committed, last_committed	MONITOR -> VALUE_VERSION	The first and last value version committed by each monitor.
num_last	MONITOR -> Int	The number of peers that accepted a collect request.
peer_first_committed, peer_last_committed	MONITOR -> (MONITOR -> VALUE_VERSION)	Used by leader when receiving responses in collect phase.
acked_lease	MONITOR -> (MONITOR -> Bool)	Stores which of the peers have acked the lease request.
pending_proposal	MONITOR -> VALUE	The value proposed by a client.
new_value	MONITOR -> VALUE	The value to be committed in the begin phase.
accepted	MONITOR -> (MONITOR -> Bool)	Stores which of the peers have acked the begin request.

3.2.3 Message Manipulation

The main way for the system to make progress is by sending and receiving messages. Therefore, it is important to have a well-defined mechanism for message exchange. In the previous section, the structure of a message and the variables that hold the messages were explained. In this section it is explained how operations such as sending and replying to messages are done.

In Listing 3.2 are some operators used for message manipulation. Namely, the operators for sending a message and replying to one recipient (lines 12 and 29), sending a message to multiple recipients (line 17), and discarding a message (line 29). These operators work by changing the value of the variable messages in the next state.

Sending a message from a monitor `mon1` to a monitor `mon2` is done by adding the message to the queue of the connection from `mon1` to `mon2` (line 2). Because messages are handled in order, removing a message is done by removing the first message in the queue of the respective connection (line 7).

```
1  \* @type: (MESSAGE, MESSAGE_QUEUE) => MESSAGE_QUEUE;
2  WithMessage(m, msgs) ==
3    [msgs EXCEPT ![m.from] =
4        [msgs[m.from] EXCEPT ![m.dest] = Append(msgs[m.from][m.dest], m)]]
5
6  \* @type: (MESSAGE, MESSAGE_QUEUE) => MESSAGE_QUEUE;
7  WithoutMessage(m, msgs) ==
8    [msgs EXCEPT ![m.from] =
9        [msgs[m.from] EXCEPT ![m.dest] = Tail(msgs[m.from][m.dest])]]
10
11 \* @type: MESSAGE => Bool;
12 Send(m) ==
13   /\ messages' = WithMessage(m, messages)
14   /\ message_history' = message_history \union {m}
15
16 \* @type: (MONITOR, Set(MESSAGE)) => Bool;
17 Send_set(from, m_set) ==
18   /\ messages' = [messages EXCEPT ![from] =
19       [mon \in Monitors |-> messages[from][mon] \o
20           SingleMessageSetToSeq({m \in m_set: m.dest = mon})]]
21   /\ message_history' = message_history \union m_set
22
23 \* @type: (MESSAGE, MESSAGE) => Bool;
24 Reply(response, request) ==
25   /\ messages' = WithoutMessage(request, WithMessage(response, messages))
26   /\ message_history' = message_history \union {response}
27
28 \* @type: MESSAGE => Bool;
29 Discard(m) ==
30   messages' = WithoutMessage(m, messages) /\ UNCHANGED message_history
```

Listing 3.2: Message manipulation operators.

3.2.4 Transitions

In this section the main transitions in the specification are explained. The transitions are divided into subsections, one for each phase of the system. First the initial and next statement of the specification is explained. Then the specification for each phase of the algorithm is presented.

3.2.4.1 Initial and Next Statement

Two of the main components in a TLA$^+$ specification are the initial and next statement. The initial statement has the definition of the initial state (or states) of the system. The next statement has the definition of the possible transitions the system can take in each action.

In the specification we define one initial state for the system. In this state, the system is in an election phase (epoch is 1), where no leader has been chosen. In the initial state, all the monitors in cluster are online, with no previous messages exchanged, and no value previously committed. The variables have default values such as the value `Nil`, defined in the constants.

```
1  Next ==
2    /\ reduce_search_space
3    /\ IF epoch % 2 = 1 THEN leader_election /\ step_name' = "election"
4       ELSE
5         \/ /\ \E mon \in Monitors: election_recover(mon)
6            /\ step_name' = "election_recover"
7         \/ /\ \E mon \in Monitors: send_collect(mon)
8            /\ step_name' = "send_collect"
9         \/ /\ \E mon \in Monitors: post_last(mon)
10           /\ step_name' = "post_last"
11        \/ /\ \E mon \in Monitors: post_lease_ack(mon)
12           /\ step_name' = "post_lease_ack"
13        \/ /\ \E mon \in Monitors: post_accept(mon)
14           /\ step_name' = "post_accept"
15        \/ /\ \E mon \in Monitors: finish_commit(mon)
16           /\ step_name' = "finish_commit"
17        \/ /\ \E mon \in Monitors: \E v \in Value_set: client_request(mon, v)
18           /\ step_name' = "client_request"
19        \/ /\ \E mon \in Monitors: propose_pending(mon)
20           /\ step_name' = "propose_pending"
21        \/ /\ \E mon1, mon2 \in Monitors:
22              /\ mon1 # mon2 /\ Len(messages[mon1][mon2])>0
23              /\ Receive(messages[mon1][mon2][1])
24        \/ /\ \E mon \in Monitors: crash_mon(mon)
25           /\ step_name' = "crash_mon"
26        \/ /\ \E mon \in Monitors: restore_mon(mon)
27           /\ step_name' = "restore_mon"
28        \/ /\ \E mon \in Monitors: Timeout(mon)
29           /\ step_name' = "timeout_and_restart"
```

Listing 3.3: Next statement.

The possible transitions the system can take, in each action, are defined in the next statement, Listing 3.3. The system executes the next statement multiple consecutive times until it reaches a state where there is no action possible (none of the actions are enabled by the current value of the variables).

The first condition in the next statement, reduce search space (line 2), evaluates if TLA$^+$ should continue to generate states. This is done to have a finite number of states to check, because the specification defines a system with an infinite number of states. The definition of this condition is discussed in section 3.3.1.

The system will then take one of the following steps, depending on the value of the variables in the current state. If the system is in an election phase, the system will elect a leader and increment the epoch (line 3). Else it will take one of the actions presented in the disjunction. Not all actions may be possible because each action has its enabling conditions (for example the system will take the action of receiving a message if there is a message to be received).

To handle the messages, there is an operator that will dispatch a message, Listing 3.4. The dispatcher will call the respective operator to handle the message, depending on the message type.

```
1  \* @type: MESSAGE => Bool;
2  Receive(msg) ==
3    \/ /\ msg.type = OP_COLLECT
4       /\ handle_collect(msg.dest, msg)   /\ step_name' = "receive collect"
5    \/ /\ msg.type = OP_LAST
6       /\ handle_last(msg.dest, msg)      /\ step_name' = "receive last"
7    \/ /\ msg.type = OP_LEASE
8       /\ handle_lease(msg.dest, msg)     /\ step_name' = "receive lease"
9    \/ /\ msg.type = OP_LEASE_ACK
10      /\ handle_lease_ack(msg.dest, msg) /\ step_name' = "receive lease_ack"
11   \/ /\ msg.type = OP_BEGIN
12      /\ handle_begin(msg.dest, msg)     /\ step_name' = "receive begin"
13   \/ /\ msg.type = OP_ACCEPT
14      /\ handle_accept(msg.dest, msg)    /\ step_name' = "receive accept"
15   \/ /\ msg.type = OP_COMMIT
16      /\ handle_commit(msg.dest, msg)    /\ step_name' = "receive commit"
```

Listing 3.4: Message dispatcher.

3.2.4.2 Leader Election and Timeouts

The leader election is the first part of the algorithm, which is executed at the start of each epoch. In the leader election, the leader for the current epoch is chosen (lines 2 to 6) and some variables are reset (lines 7 to 11) to a default value (e.g. the messages in the network are cleared). The leader chosen in the election will lead all the commits in that epoch. In Listing 3.5 is the definition of the election operator.

The quorum defined in the election has all the current not crashed monitors, and the leader is chosen from that set. If the quorum is of size one, the system will enter an idle state waiting for client request (active state). Else the system will enter a recovering state where the monitors will synchronize their values (recovering state). The recovering phase starts by executing the election recover operator that will trigger a collect phase (section 3.2.4.3).

```
1  leader_election ==
2    /\ \E mon \in Monitors:
3        /\ quorum[mon]
4        /\ isLeader' = [m \in Monitors |-> IF m = mon THEN TRUE ELSE FALSE]
5        /\ state' = [m \in Monitors |->
6            IF quorum_sz = 1 THEN STATE_ACTIVE ELSE STATE_RECOVERING]
7    /\ phase' = [m \in Monitors |-> PHASE_ELECTION]
8    /\ new_value' = [m \in Monitors |-> Nil]
9    /\ pending_proposal' = [m \in Monitors |-> Nil]
10   /\ epoch' = epoch + 1
11   /\ messages' = [mon1 \in Monitors |-> [mon2 \in Monitors |-> <<>>] ]
12   /\ UNCHANGED <<quorum, quorum_sz, accepted, message_history>>
13   /\ UNCHANGED <<data_vars, restart_vars, collect_vars, lease_vars>>
```

Listing 3.5: Leader election.

An election is triggered when the epoch is odd. This happens at the start of the system and whenever a monitor crashes, recovers, or has timeout. In Listing 3.6 are the operators used to define a crash (line 2) and a recover (line 12) of a monitor. This is done by changing the value of the quorum variable (lines 5 and 14).

```
1  \* @type: MONITOR => Bool;
2  crash_mon(mon) ==
3    /\ quorum_sz > (MonitorsLen \div 2) + 1
4    /\ quorum[mon] = TRUE
5    /\ quorum' = [quorum EXCEPT ![mon] = FALSE]
6    /\ quorum_sz' = quorum_sz - 1
7    /\ bootstrap
8    /\ UNCHANGED <<messages, message_history>>
9    /\ UNCHANGED <<state_vars, restart_vars, data_vars, collect_vars, lease_vars,
        commit_vars>>
10
11 \* @type: MONITOR => Bool;
12 restore_mon(mon) ==
13   /\ quorum[mon] = FALSE
14   /\ quorum' = [quorum EXCEPT ![mon] = TRUE]
15   /\ quorum_sz' = quorum_sz + 1
16   /\ bootstrap
17   /\ UNCHANGED <<messages, message_history>>
18   /\ UNCHANGED <<state_vars, restart_vars, data_vars, collect_vars, lease_vars,
        commit_vars>>
```

Listing 3.6: Definition of the operators for a monitor crash and recover.

A crash of a monitor can occur if the number of monitors in the quorum is bigger than half of the total number of monitors (line 3). This is because the algorithm does not tolerate arbitrary number of failures, to tolerate f failures it requires a cluster size of at least $2f + 1$ monitors. As stated before, both the recovering and crash of a monitor will trigger new elections (lines 7 and 16), by incrementing the epoch (defined in the bootstrap operator).

3.2.4.3 Collect Phase

The collect phase is when the monitors in the quorum synchronize their values (it can also be referred to as the recovering phase). This phase occurs at the start of each epoch, if the number of monitors in the cluster is bigger than one (if there is only one monitor it does not have anyone to synchronize with). In Listing 3.7 are the operators used to start a collect phase.

```
1  \* @type: (MONITOR, Int) => Bool;
2  collect(mon, oldpn) ==
3    /\ isLeader[mon] = TRUE
4    /\ state[mon] = STATE_RECOVERING
5    /\ LET new_pn == get_new_proposal_number(mon, Max({oldpn,accepted_pn[mon]}))
6       IN  /\ accepted_pn' = [accepted_pn EXCEPT ![mon] = new_pn]
7    /\ phase' = [phase EXCEPT ![mon] = PHASE_SEND_COLLECT]
8
9  \* @type: MONITOR => Bool;
10 send_collect(mon) ==
11   /\ isLeader[mon] = TRUE
12   /\ state[mon] = STATE_RECOVERING
13   /\ phase[mon] = PHASE_SEND_COLLECT
14   /\ clear_peer_first_committed(mon) /\ clear_peer_last_committed(mon)
15   /\ IF last_committed[mon]+1 \in DOMAIN values[mon]
16      THEN /\ uncommitted_v' = [uncommitted_v EXCEPT ![mon] =
17                                last_committed[mon]+1 ]
18           /\ uncommitted_value' = [uncommitted_value EXCEPT ![mon] =
19                                    values[mon][last_committed[mon]+1] ]
20           /\ uncommitted_pn' = [uncommitted_pn EXCEPT ![mon] = pending_pn[mon]]
21           /\ UNCHANGED <<pending_pn, pending_v>>
22      ELSE UNCHANGED <<restart_vars>>
23   /\ num_last' = [num_last EXCEPT ![mon] = 1]
24   /\ Send_set(mon, {[type            |-> OP_COLLECT,
25                      from            |-> mon,
26                      dest            |-> dest,
27                      first_committed |-> first_committed[mon],
28                      last_committed  |-> last_committed[mon],
29                      pn              |-> accepted_pn[mon]]
30                    : dest \in {m \in (Monitors \ {mon}): quorum[m]} })
31   /\ phase' = [phase EXCEPT ![mon] = PHASE_COLLECT]
32   /\ UNCHANGED <<isLeader, state>>
33   /\ UNCHANGED <<epoch, quorum, quorum_sz, data_vars, lease_vars, commit_vars>>
```

Listing 3.7: Operators used to start a collect phase.

In the collect phase, the monitors use proposal numbers to synchronize themselves, similar to the proposal numbers in paxos. These numbers are computed using a strictly increasing function (Listing 3.8). The function used in the specification is slightly different from the one in the implementation. In the implementation, instead of the epoch, they use an identifier of the monitor (the monitor rank), however this makes the monitor set not symmetric which makes the specification less efficient when model checking.

```
1 \* @type: (MONITOR, Int) => Int;
2 get_new_proposal_number(mon, oldpn) == ((oldpn \div 100) + 1) * 100 + epoch
```

Listing 3.8: Operator that generates proposal numbers.

The collect operator (line 2) is used to trigger a collect phase and it receives as arguments the leader monitor and the last proposal number he knows. This operator assigns the state of the monitor in the next state to sending the collect message (line 7) and saves the proposal number that will be used in `accepted_pn` (line 6).

In the send collect operator (line 10), the leader sends collect messages to all the monitors in the quorum with the new proposal number (line 24). The monitor also resets the variables that hold the first and last version that the peers have committed (that will be used later on) and the variable `num_last` that holds the number of monitors that have accpeted the new proposal number. If the leader has an uncommitted value, it will also save it in the uncommitted variables.

```
1  \* @type: (MONITOR, MESSAGE) => Bool;
2  handle_collect(mon, msg) ==
3    /\ isLeader[mon] = FALSE
4    /\ state' = [state EXCEPT ![mon] = STATE_RECOVERING]
5    /\ \/ /\ msg.first_committed > last_committed[mon] + 1
6          /\ bootstrap /\ Discard(msg)
7          /\ UNCHANGED <<accepted_pn>>
8       \/ /\ msg.first_committed <= last_committed[mon] + 1
9          /\ IF msg.pn > accepted_pn[mon]
10            THEN accepted_pn' = [accepted_pn EXCEPT ![mon] = msg.pn]
11            ELSE UNCHANGED accepted_pn
12         /\ Reply([type            |-> OP_LAST,
13                   from            |-> mon,
14                   dest            |-> msg.from,
15                   first_committed |-> first_committed[mon],
16                   last_committed  |-> last_committed[mon],
17                   values          |-> values[mon],
18                   uncommitted_pn  |-> pending_pn[mon],
19                   pn              |-> accepted_pn'[mon]],msg)
20         /\ UNCHANGED epoch
21   /\ UNCHANGED <<quorum, quorum_sz, isLeader, phase, values, first_committed,
        last_committed, monitor_store>>
22   /\ UNCHANGED <<restart_vars, collect_vars, lease_vars, commit_vars>>
```

Listing 3.9: Operator to handle a collect message.

Listing 3.9 shows the operator used to handle collect messages. When a peer receives a collect message, it first checks if the leader is consistent (if the first commit of the leader is smaller than the last commit of the peer, line 5), and if not, it triggers new elections (line 6). If it is consistent, the peer will reply with a last message with a proposal number obtained from the maximum between the one he has and the one in the collect message (line 12).

```
1  \* @type: (MONITOR, MESSAGE) => Bool;
2  handle_last(mon,msg) ==
3    /\ isLeader[mon] = TRUE
4    (update peer_first_committed and peer_last_committed)
5    /\ IF msg.first_committed > last_committed[mon] + 1
6       THEN
7        /\ bootstrap
8       ELSE
9        /\ store_state(mon, msg)
10       /\ IF \E peer \in Monitors:
11               /\ peer # mon
12               /\ peer_last_committed'[mon][peer] # -1
13               /\ peer_last_committed'[mon][peer] + 1 < first_committed[mon]
14               /\ first_committed[mon] > 1
15          THEN
16           /\ bootstrap
17          ELSE
18           /\ LET monitors_behind == {peer \in Monitors:
19                   /\ peer # mon
20                   /\ peer_last_committed'[mon][peer] # -1
21                   /\ peer_last_committed'[mon][peer] < last_committed[mon]
22                   /\ quorum[peer]}
23              IN Reply_set(mon,
24                   {[type           |-> OP_COMMIT,
25                     from           |-> mon,
26                     dest           |-> dest,
27                     last_committed |-> last_committed'[mon],
28                     pn             |-> accepted_pn[mon],
29                     values         |-> values[mon]]: dest \in monitors_behind
30                   }, msg)
31           /\ \/ /\ msg.pn > accepted_pn[mon]
32                 /\ collect(mon, msg.pn)
33              \/ /\ msg.pn = accepted_pn[mon]
34                 /\ num_last' = [num_last EXCEPT ![mon] = num_last[mon] + 1]
35                 /\ IF /\ msg.last_committed+1 \in DOMAIN msg.values
36                       /\ msg.last_committed >= last_committed'[mon]
37                       /\ msg.last_committed+1 >= uncommitted_v[mon]
38                       /\ msg.uncommitted_pn >= uncommitted_pn[mon]
39                    THEN /\ (update uncommitted_<v, pn, value>)
40                    ELSE check_and_correct_uncommitted(mon)
41   (...)
```

Listing 3.10: Excerpt of the operator to handle a last message.

Listing 3.10 shows an excerpt of the operator used by the leader to handle a last message from his peers. This is one of the most important operators because it is here that the leader checks if there are uncommitted values from previous commit phases that were stopped. In the previous commit phases, some monitors may have committed a value, and it was the case, the leader must propose that value again to prevent different commit values in different monitors.

The leader starts by updating some of his variables and then checks if the system is consistent (line 10), if not he triggers new elections. If the system is consistent, the leader checks if there are monitors behind (with an incomplete log, line 18) and sends them commit messages with his own log for the peers to update (line 23).

Then the leader will take one of three actions depending on the proposal number on the message. If the proposal number in the message is bigger than the one that he has, the leader starts a new collect phase with a new proposal number (line 31). If the proposal number is smaller, he ignores the message. Finally, if it is the same proposal number, the leader increments the `num_last` variable (the peer accepted the collect request) and updates the uncommitted variables with the value from the peers, if he has a more recent uncommitted value than the one the leader knows (line 33).

```
1  \* @type: MONITOR => Bool;
2  post_last(mon) ==
3    /\ isLeader[mon] = TRUE /\ phase[mon] = PHASE_COLLECT
4    /\ num_last[mon] = quorum_sz
5    /\ clear_peer_first_committed(mon)
6    /\ clear_peer_last_committed(mon)
7    /\ IF /\ uncommitted_v[mon] = last_committed[mon]+1
8          /\ uncommitted_value[mon] # Nil
9       THEN /\ state' = [state EXCEPT ![mon] = STATE_UPDATING_PREVIOUS]
10           /\ begin(mon, uncommitted_value[mon])
11           /\ UNCHANGED <<acked_lease, uncommitted_v, uncommitted_pn,
                uncommitted_value>>
12      ELSE /\ finish_round(mon) /\ extend_lease(mon)
13           /\ UNCHANGED <<accepted, new_value, values, restart_vars>>
14   /\ UNCHANGED <<isLeader, monitor_store, accepted_pn, first_committed,
        last_committed>>
15   /\ UNCHANGED <<epoch, quorum, quorum_sz, num_last, pending_proposal>>
```

Listing 3.11: Operator that ends collect phase.

If the leader receives positive responses from all peers in the quorum to his collect request, he calls the post last operator (Listing 3.11). It there is an uncommitted value the leader starts a new commit phase with that value (line 10). Else, the leader starts a lease phase, which transitions the system to an idle state where the leader waits for client requests (line 12). The lease phase also enables the clients to read the system state from any monitor in the quorum.

3.2.4.4 Commit Phase

The commit phase is when the leader tries to commit a new version of a value to the storage. The commit phase can be triggered after a collect phase, when there is an uncommitted value, or by a client request. The operators used to handle a client request and then start a commit phase are shown in Listing 3.12.

```
1  \* @type: (MONITOR, VALUE) => Bool;
2  client_request(mon, v) ==
3    /\ isLeader[mon] = TRUE
4    /\ state[mon] = STATE_ACTIVE
5    /\ pending_proposal[mon] = Nil
6    /\ pending_proposal' = [pending_proposal EXCEPT ![mon] = v]
7    /\ UNCHANGED <<global_vars, state_vars, new_value, accepted>>
8    /\ UNCHANGED <<restart_vars, data_vars, collect_vars, lease_vars>>
9
10 \* @type: MONITOR => Bool;
11 propose_pending(mon) ==
12   /\ phase[mon] = PHASE_LEASE \/ phase[mon] = PHASE_ELECTION
13   /\ state[mon] = STATE_ACTIVE
14   /\ pending_proposal[mon] # Nil
15   /\ pending_proposal' = [pending_proposal EXCEPT ![mon] = Nil]
16   /\ state' = [state EXCEPT ![mon] = STATE_UPDATING]
17   /\ begin(mon, pending_proposal[mon])
18   /\ UNCHANGED <<accepted_pn, first_committed, last_committed, epoch, quorum,
         quorum_sz, uncommitted_v, uncommitted_pn, uncommitted_value>>
19   /\ UNCHANGED <<isLeader, monitor_store, collect_vars, lease_vars>>
```

Listing 3.12: Operators that handles a client request.

The first operator, in line 2, is used to add the new requested value to the `pending_proposal` variable of the leader. The second operator, in line 11, is used to start a commit phase with the value in pending proposal. In the implementation, the client request can be done to any monitor (and the peers redirect it to the leader), and at any state. In the specification, it was restricted to be called only by the leader when he is in an active state, because it had a significant impact on performance, and it does not impact the safety analysis.

The operator that starts the commit phase is the begin operator (shown in Listing 3.13). This operator is called in the `post_last` and `propose_pending` operators, which were previously presented. In this operator, the leader sends begin messages to all the monitors in the quorum, informing them of the new value the leader is proposing (line 13). The leader also saves the new value in the log (line 11), and in the pending variables (lines 21 and 22). This is done in order for the leader to be able to retrieve the value in case the commit phase is stopped.

Note that the variable values has values that are not committed. The values from version `first_committed` to version `last_committed` are values that were indeed committed, and remaining ones are uncommitted values.

```
1  \* @type: (MONITOR, VALUE) => Bool;
2  begin(mon, v) ==
3    /\ isLeader[mon] = TRUE
4    /\ state'[mon] = STATE_UPDATING \/ state'[mon] = STATE_UPDATING_PREVIOUS
5    /\ quorum_sz = 1 \/ num_last[mon] > MonitorsLen \div 2
6    /\ new_value[mon] = Nil
7    /\ accepted' = [accepted EXCEPT ![mon] =
8          [m \in Monitors |-> IF m = mon THEN TRUE ELSE FALSE]]
9    /\ new_value' = [new_value EXCEPT ![mon] = v]
10   /\ phase' = [phase EXCEPT ![mon] = PHASE_BEGIN]
11   /\ values' = [values EXCEPT ![mon] =
12         ((last_committed[mon] + 1) :> new_value'[mon]) @@ values[mon] ]
13   /\ Send_set(mon,
14         {[type           |-> OP_BEGIN,
15           from           |-> mon,
16           dest           |-> dest,
17           last_committed |-> last_committed[mon],
18           values         |-> values'[mon],
19           pn             |-> accepted_pn[mon]]
20         : dest \in {m \in Monitors \ {mon}: quorum[m]} })
21   /\ pending_pn' = [pending_pn EXCEPT ![mon] = accepted_pn[mon]]
22   /\ pending_v' = [pending_v EXCEPT ![mon] = last_committed[mon]+1]
```

Listing 3.13: Operators to handle a client request.

Listing 3.14 depicts an excerpt (without unchanged statements) of the operator used to handle begin messages. When receiving a begin message, a monitor will accept it if it has the same proposal number that he previously accepted (line 4), and if the proposer has an updated log (same last committed version, line 7). If the monitor accepts the begin request, he replies with an accept message (line 13) and saves the new value in the log and in the pending variables.

```
1  \* @type: (MONITOR, MESSAGE) => Bool;
2  handle_begin(mon, msg) ==
3    /\ isLeader[mon] = FALSE
4    /\ IF msg.pn < accepted_pn[mon]
5       THEN /\ Discard(msg)
6       ELSE
7         /\ msg.pn = accepted_pn[mon] /\ msg.last_committed = last_committed[mon]
8         /\ values' = [values EXCEPT ![mon] = ((last_committed[mon]+1) :>
9                       msg.values[last_committed[mon]+1]) @@ values[mon] ]
10        /\ state' = [state EXCEPT ![mon] = STATE_UPDATING]
11        /\ pending_pn' = [pending_pn EXCEPT ![mon] = accepted_pn[mon]]
12        /\ pending_v' = [pending_v EXCEPT ![mon] = last_committed[mon]+1]
13        /\ Reply([type             |-> OP_ACCEPT,
14                  from             |-> mon,
15                  dest             |-> msg.from,
16                  last_committed   |-> last_committed[mon],
17                  pn               |-> accepted_pn[mon]],msg)
```

Listing 3.14: Excerpt of the operator to handle a begin message.

In Listing 3.15 is shown the operator used by the leader to handle accept messages. If the message is referent to the leader begin request, the leader saves that the monitor accepted the request, in the `accept` variable (line 10).

```
\* @type: (MONITOR, MESSAGE) => Bool;
handle_accept(mon, msg) ==
  /\ isLeader[mon] = TRUE
  /\ state[mon] = STATE_UPDATING_PREVIOUS \/ state[mon] = STATE_UPDATING
  /\ phase[mon] = PHASE_BEGIN
  /\ new_value[mon] # Nil
  /\ IF \/ msg.pn # accepted_pn[mon]
        \/ last_committed[mon] > 0 /\ msg.last_committed < last_committed[mon]-1
     THEN UNCHANGED accepted
     ELSE accepted' = [accepted EXCEPT ![mon] =
              [accepted[mon] EXCEPT ![msg.from] = TRUE]]
  /\ Discard(msg)
```

Listing 3.15: Excerpt of the operator to handle an accept message.

When all the monitors in the quorum accept the begin request, the leader calls the `post_accept` operator, Listing 3.16. In this operator, the leader commits locally the value that was proposed (lines 6 and 11) and sends commit messages to the monitors in quorum (line 14), allowing them to commit the new value as well. The value is committed by changing the value of the `monitor_store` variable and incrementing the variable with the last committed version.

```
\* @type: MONITOR => Bool;
post_accept(mon) ==
  /\ \A m \in Monitors: quorum[m] => accepted[mon][m] = TRUE
  /\ new_value[mon] # Nil /\ phase[mon] = PHASE_BEGIN
  /\ state[mon] = STATE_UPDATING_PREVIOUS \/ state[mon] = STATE_UPDATING
  /\ last_committed' = [last_committed EXCEPT ![mon] = last_committed[mon] + 1]
  /\ IF first_committed[mon] = 0
     THEN first_committed' =
            [first_committed EXCEPT ![mon] = first_committed[mon] + 1]
     ELSE UNCHANGED first_committed
  /\ monitor_store' =
       [monitor_store EXCEPT ![mon] = values[mon][last_committed[mon]+1]]
  /\ new_value' = [new_value EXCEPT ![mon] = Nil]
  /\ Send_set(mon,
       {[type           |-> OP_COMMIT,
         from           |-> mon,
         dest           |-> dest,
         last_committed |-> last_committed'[mon],
         pn             |-> accepted_pn[mon],
         values         |-> values[mon]]
       : dest \in {m \in Monitors \ {mon}: quorum[m]} })
  /\ state' = [state EXCEPT ![mon] = STATE_REFRESH]
  /\ phase' = [phase EXCEPT ![mon] = PHASE_COMMIT]
```

Listing 3.16: Excerpt of the operator to send the commit messages.

When a monitor receives a commit message, it commits all values that are shared in the message. This synchronization is done using the operator shown in Listing 3.17. After sending the commit messages, the leader starts a lease phase to transition the system to an idle state. To trigger the lease phase the leader calls the `finish_commit` operator.

```
1  \* @type: (MONITOR, MESSAGE) => Bool;
2  store_state(mon,msg) ==
3    /\ LET logs == (DOMAIN msg.values) \intersect
         (last_committed[mon]+1..msg.last_committed)
4       IN /\ values' = [values EXCEPT ![mon] =
5                 [i \in DOMAIN values[mon] \union logs |->
6                     IF i \in logs
7                     THEN msg.values[i]
8                     ELSE values[mon][i] ]]
9          /\ last_committed' = [last_committed EXCEPT ![mon] =
10                                Max(logs \union {last_committed[mon]})]
11         /\ IF logs # {} /\ first_committed[mon] = 0
12            THEN first_committed' =
13                      [first_committed EXCEPT ![mon] = Min(logs)]
14            ELSE first_committed' = [first_committed EXCEPT ![mon] =
15                                      Min(logs \union {first_committed[mon]})]
16   /\ IF last_committed'[mon] = 0
17      THEN UNCHANGED monitor_store
18      ELSE monitor_store' = [monitor_store EXCEPT ![mon] =
19                              values'[mon][last_committed'[mon]]]
```

Listing 3.17: Operator used to commit values shared in a commit message.

3.2.4.5 Lease Phase

The lease phase is used to enable the clients to read the value in the storage of the monitors in the quorum. This is done by transition the system to an idle state, where the monitors are in the state named `STATE_ACTIVE`. The goal of the lease is to give guarantees to the clients that the value they are reading is the most updated one in the quorum. The lease is also used as a heartbeat to check if the peers of a monitor are still online.

The complete lease phase is written in the specification, however only a part of it is used. The heartbeat part of the lease phase was disabled in the specification because it added a lot of overhead in the model checker. The timeout is specified in a simpler way using the timeout operator, which increments the epoch and triggers new elections.

To start a lease phase, the leader uses the `finish_round` and `extend_lease` operator, Listing 3.18. With the first operator the leader changes its state to `STATE_ACTIVE` (line 4), which enables reads from the clients. With the second operator, the leader sends lease messages to the peers instructing them to change their states to `STATE_ACTIVE` (line 11). In the implementation, leases have timeouts, and the leader has to periodically send lease messages, which is not the

case in the specification.

```
1  \* @type: MONITOR => Bool;
2  finish_round(mon) ==
3      /\ isLeader[mon] = TRUE
4      /\ state' = [state EXCEPT ![mon] = STATE_ACTIVE]
5
6  \* @type: MONITOR => Bool;
7  extend_lease(mon) ==
8    /\ isLeader[mon] = TRUE
9    /\ acked_lease' = [acked_lease EXCEPT ![mon] =
10       [m \in Monitors |-> IF m = mon THEN TRUE ELSE FALSE]]
11   /\ Send_set(mon,
12       {[type           |-> OP_LEASE,
13         from           |-> mon,
14         dest           |-> dest,
15         last_committed |-> last_committed[mon]]: dest \in {m \in Monitors \
             {mon}: quorum[m]}
16       })
17   /\ phase' = [phase EXCEPT ![mon] = PHASE_LEASE]
```

Listing 3.18: Operators used to start a lease phase.

Listing 3.19 depicts the operator used by a monitor to handle a lease message. When a monitor receives a lease message, it will change its state to `STATE_ACTIVE` (line 8). In the implementation the monitor would reply with a lease ack message (line 9), however that is commented in the specification because of the impact in the model checker and that behaviour is instead specified using the timeout operator. In the implementation, the leader adds the monitors that reply with a lease ack message to the `acked_lease` variable. If the leader does not receive replies from all monitors in the quorum, he triggers new elections.

```
1  \* @type: (MONITOR, MESSAGE) => Bool;
2  handle_lease(mon, msg) ==
3    /\ \* discard if not peon or peon is behind
4       IF \/ isLeader[mon] = TRUE
5          \/ last_committed[mon] # msg.last_committed
6       THEN /\ Discard(msg)
7            /\ UNCHANGED state
8       ELSE /\ state' = [state EXCEPT ![mon] = STATE_ACTIVE]
9            (*/\ Reply([type            |-> OP_LEASE_ACK,
10                       from            |-> mon,
11                       dest            |-> msg.from,
12                       first_committed |-> first_committed[mon],
13                       last_committed  |-> last_committed[mon]],msg)*)
14           /\ Discard(msg)
15   /\ UNCHANGED <<epoch, quorum, quorum_sz, isLeader, phase>>
16   /\ UNCHANGED <<restart_vars, data_vars, collect_vars, lease_vars,
        commit_vars>>
```

Listing 3.19: Operator to handle a lease message.

3.3 Verifying the Safety of the Algorithm

The specification presented in the previous section can be checked for safety using a model checker. The model checker chosen was TLC. It checks a specification by enumerating all possible states and checking if these states are consistent, using a definition of consistency provided by the user. This section presents how the search space was limited and the safety properties that were defined.

3.3.1 Limiting the Space Exploration

The specification that was presented has an infinite number of states because values can continuously be proposed and committed, and new elections can continuously take place. However, the model checker is only able to check a finite number of states. Therefore, in order to be able to check the specification, the search space has to be limited.

Listing 3.20 depicts the operator used to limit the search space. This operator is used in every transition to check if the new state is inside the search space. The variables used to limit search space are: epoch, to limit the number of elections that can occur; `last_committed`, to limit the number of values that can be committed; and `accepted_pn`, to limit the number of a collect phases.

```
1 reduce_search_space ==
2   /\ epoch # 14
3   /\ \A mon \in Monitors: last_committed[mon] < 2
4   /\ \A mon \in Monitors: accepted_pn[mon] < 400
```

Listing 3.20: Operator to limit the search space.

The values chosen to limit the search space were obtained by introducing errors in the specification and checking how large the search space needed to be to find those errors. An example is presented in the Section Testing the Specification (4.1).

3.3.2 Invariants

In order for the model checker to evaluate the safety of a specification, the user needs to provide the properties that the system must have. TLC can check two kinds of properties: invariants, statements that are true in every state; and temporal formulas, statements that are true in behaviours (a sequence of states).

A temporal formula is for testing temporal properties (liveness), for example a property like: if there is a state X then eventually a state Y is reached. An invariant is for testing safety properties. This work focuses only on invariants, which are explained in more detail below.

```
1  same_monitor_store ==
2    \A mon1, mon2 \in Monitors:
3      state[mon1] = STATE_ACTIVE /\ state[mon2] = STATE_ACTIVE
4      => monitor_store[mon1] = monitor_store[mon2]
5
6  same_monitor_values ==
7    \A version \in 1..Max({last_committed[mon]: mon \in Monitors}):
8      \E val \in Value_set:
9        \A mon \in Monitors:
10         last_committed[mon] < version \/ values[mon][version] = val
```

Listing 3.21: Safety invariants.

Listing 3.21 depicts the two invariants used in this work. The first invariant, in line 1, states that, if two monitors are in the state `STATE_ACTIVE`, then the value in their store is the same. If the monitors do not have the same value, then different clients could read different values, depending on which monitor they chose to read it from, making the system inconsistent.

The second invariant, in line 6, states that all the monitors make the same value decision in each version. This statement is the traditional safety property for consensus, where only one value is chosen for each version.

3.4 Algorithm Visualization

Two important ways of studying and analysing a TLA$^+$ specification are state graphs and trace errors. However, both of them can be hard to analyse using traditional tools. This section introduces a novel tool created to explore and analyse both state graphs and trace errors.

3.4.1 TLA$^+$ Graph and Trace Explorer

State graphs and traces of a specification can be created using the TLC model checker. The state graphs are created using the dot format and the traces in a text format. Both of them can be hard to analyse when studying a complex specification because each state tends to be complex and large. State graphs also tend to be too big to be able to analyse using traditional graph explorers.

In Figure 3.5 is an example of a state graph, where each state has a complete description of the variables. A state graph of a complex specification can have thousands of states, making the graphs hard to visualize and study. The tools also tend to have low performance when rendering big graphs. In Figure 3.6 is an example of a trace file, where each step of the trace has a description of the state variables.

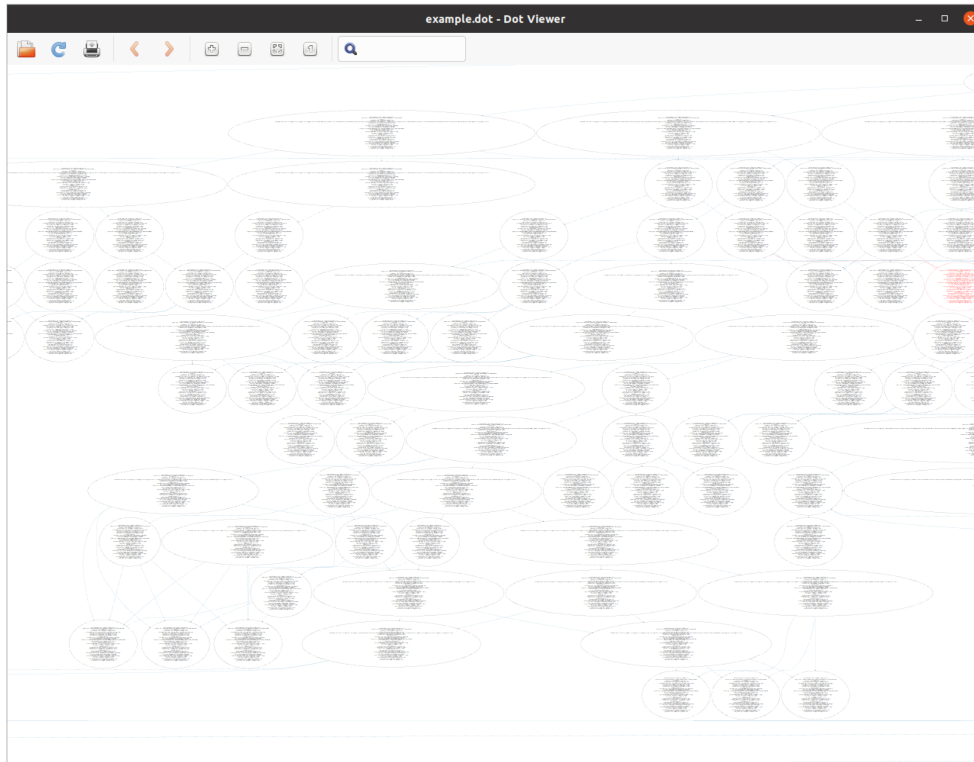


Figure 3.5: Visualization of a state graph using a traditional tool (xdot[4]).

trace.txt
~/sandbox/reps/ceph-consensus-spec/animation

```
1  <Initial predicate>
2         /\ uncommitted_v = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
3         /\ phase = (m1 :> PHASE_ELECTION @@ m2 :> PHASE_ELECTION @@ m3 :> PHASE_ELECTION)
4         /\ epoch = 1
5         /\ pending_v = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
6         /\ monitor_store = (m1 :> Nil @@ m2 :> Nil @@ m3 :> Nil)
7         /\ messages = ( m1 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) @@
8   m2 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) @@
9   m3 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) )
10        /\ accepted_pn = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
11        /\ isLeader = (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE)
12        /\ number_crashes = 0
13        /\ accepted = ( m1 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) @@
14  m2 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) @@
15  m3 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) )
16        /\ state = (m1 :> STATE_RECOVERING @@ m2 :> STATE_RECOVERING @@ m3 :> STATE_RECOVERING)
17        /\ first_committed = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
18        /\ step_name = "init"
19        /\ pending_pn = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
20        /\ peer_first_committed = ( m1 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) @@
21  m2 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) @@
22  m3 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) )
23        /\ quorum_sz = 3
24        /\ pending_proposal = (m1 :> Nil @@ m2 :> Nil @@ m3 :> Nil)
25        /\ values = (m1 :> << >> @@ m2 :> << >> @@ m3 :> << >>)
26        /\ quorum = (m1 :> TRUE @@ m2 :> TRUE @@ m3 :> TRUE)
27        /\ message_history = {}
28        /\ last_committed = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
29        /\ uncommitted_pn = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
30        /\ acked_lease = ( m1 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) @@
31  m2 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) @@
32  m3 :> (m1 :> FALSE @@ m2 :> FALSE @@ m3 :> FALSE) )
33        /\ peer_last_committed = ( m1 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) @@
34  m2 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) @@
35  m3 :> (m1 :> -1 @@ m2 :> -1 @@ m3 :> -1) )
36        /\ uncommitted_value = (m1 :> Nil @@ m2 :> Nil @@ m3 :> Nil)
37        /\ num_last = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
38        /\ new_value = (m1 :> Nil @@ m2 :> Nil @@ m3 :> Nil)
39
40 <Next line 1022, col 5 to line 1076, col 38 of module ceph>
41        /\ uncommitted_v = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
42        /\ phase = (m1 :> PHASE_ELECTION @@ m2 :> PHASE_ELECTION @@ m3 :> PHASE_ELECTION)
43        /\ epoch = 2
44        /\ pending_v = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
45        /\ monitor_store = (m1 :> Nil @@ m2 :> Nil @@ m3 :> Nil)
46        /\ messages = ( m1 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) @@
47  m2 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) @@
48  m3 :> (m1 :> <<>> @@ m2 :> <<>> @@ m3 :> <<>>) )
49        /\ accepted_pn = (m1 :> 0 @@ m2 :> 0 @@ m3 :> 0)
50        /\ isLeader = (m1 :> TRUE @@ m2 :> FALSE @@ m3 :> FALSE)
51        /\ number_crashes = 0
```

Plain Text Tab Width: 8 Ln 73, Col 46 INS

Figure 3.6: Trace error file example.

[4] https://github.com/jrfonseca/xdot.py

In the context of this work, a new tool was developed to make it easier to study a specification. The tool is named TLA$^+$ Graph Explorer and supports a novel way to visualize state graphs and trace errors. The tool is open-sourced and hosted on GitHub[5].

The main goal of the tool was to make state graphs easier to study and analyse. By studying a state graph, the user can see what states are being generated and what transitions are possible in each state. This information helps to understand how the system works and find errors in the specification (transitions that should not be there).

The main challenge faced when developing the tool was the file size of state graphs, which can easily reach multiple gigabytes of data. Because of this, the information of the graph has to be efficiently stored in memory. One of the main issues when using tools that rendered graphs was that they were very slow and, many times, could not even render the graph because of its size.

Therefore, the tool focuses only on rendering one state and its children at a time. This approach is better suited for a state graph of a specification as it gives better insights of how the system evolves and does not require to have the whole graph in memory, making it feasible to study big state graphs.

The complex specifications that this tool is focused on normally have complex states that can be hard to study using only the variable states printed in text format. Because of this, the tool also provides a way to personalize how a state is rendered. A personalized state helps the user to understand each state more easily and quickly. It also enables a way to create visual animations that can be shared with team members that do not know the TLA$^+$ language.

After the development of the graph explorer, a slightly different version of the tool was developed to also support trace errors, the TLA$^+$ Trace Explorer. For trace errors, there was already a way to make animations and personalize each state[6], however the task is easier using this tool.

In the Figure 3.7 is an example of the usage of the tool with the Ceph consensus algorithm specification that was previously presented. The information of a state is presented graphically, with information of the various connections between the monitors and the current state of each monitor.

The tool was developed using JavaScript. Because of this it can run in almost any environment, requiring only a browser. The files used in the tool (state graphs and trace errors) are read in chunks and only a pointer to each node is saved, reducing the amount of memory required to run the tool. The tool saves in memory the current node and its children and the path until the current node to be able to backtrack.

A user can personalize how a state is rendered using HTML, CSS, and JavaScript. The default flow is to create a html template of the structure of the state, styling it with CSS and filling the values with JavaScript. A more in-depth look is presented in the next subsection.

[5]https://github.com/afonsonf/tlaplus-graph-explorer

[6]https://github.com/tlaplus/tlaplus/issues/485#issuecomment-656423026

A parser for a TLA$^+$ expression was also developed to help the user use the state variables in JavaScript. The parser was written in Jison[7], a JavaScript version of Bison, and it converts a TLA$^+$ state to JavaScript objects.

In summary, the key features of this tool are:

- Can run in a browser.
- Interactively explore state graphs and trace errors.
- Ability to personalize how each state is rendered.
- Developed to support big state graphs.

3.4.2 Visualization of the Ceph Consensus Algorithm

In order to help the study of the Ceph consensus algorithm specification, a personalized state representation was developed using the previous tool, Figure 3.7. The personalized state helps identify what is happening in each state by using visual representations of the state variables. The same personalized state representation can be used both for visualization of a state graph and a trace error.

In the Listing 3.22 is the HTML template used to render the information about each monitor (the small rectangles at the top of the state). Afterwords, in JavaScript, the elements of the html template are filled with the values of the monitor variables.

```
1  <template id="monitorTemplate">
2    <div id="monitor-div" class="state-monitor">
3      <div style="display:flex;">
4        <p id="monitor-name" style="font-size: 16px; font-weight: 700;"></p>
5        <div id="monitor-value" style="font-size: 16px;" class="monitor-value">
6        </div>
7      </div>
8      <div> Role: <p id="monitor-role"></p> </div>
9      <div> State: <p id="monitor-state" style="font-size: 14px"></p> </div>
10     <div> Pn: <p id="monitor-pn"></p> </div>
11     <div> New value: <p id="monitor-new_value"></p> </div>
12     <div> Pending pn: <p id="monitor-pending_pn"></p> </div>
13     <div> Uncommitted: <br/>
14            pn: <p id="monitor-uncommitted_pn"></p>
15           version: <p id="monitor-uncommitted_v"></p>
16           value: <p id="monitor-uncommitted_value"></p>
17     </div>
18   </div>
19 </template>
```

Listing 3.22: HTML template to render the information of a monitor.

[7] https://github.com/zaach/jison

Figure 3.7: Usage example of the TLA$^+$ Graph Explorer.

Chapter 4

Results and Analysis

4.1 Testing the Specification

To evaluate the safety of the specification and if the specification matches the implementation, two tests are presented. In the first test a bug is introduced in the specification to assess if the specification produces the same error that is produced by the implementation. And in the second example, the specification (without any changes) is checked to verify the safety of the implemented algorithm. Both examples use the same configuration file that is presented in Appendix A.

The bug that was introduced is a bug that was present in an early version of Ceph (fixed in the commit *20baf662112dd5f560bc3a2d2114b469444c3de8* of the Ceph project[1]). In this bug, the proposal number associated with an uncommitted value is not saved (in the variable `pending_pn`). The modifications to introduce the bug in the specification are presented in Listing 4.1.

```
1 In the send_collect collect operator:
2 [-] /\ uncommitted_pn' = [uncommitted_pn EXCEPT ![mon] = pending_pn[mon]]
3 [+] /\ uncommitted_pn' = [uncommitted_pn EXCEPT ![mon] = accepted_pn[mon]]
4
5 In the handle_collect operator:
6 [-] uncommitted_pn  |-> pending_pn[mon],
7 [+] uncommitted_pn  |-> accepted_pn[mon],
```

Listing 4.1: Modifications to the specification to introduce the bug.

Running the TLC model checker with the modified specification yields an error, stating that an invariant has been violated. The output comes with a trace, which is an example of a sequence of steps that leads to an inconsistent state. The trace can be studied to see what caused the violation and fix the bug. A tool to help visualize the trace is presented in the next section.

[1] https://github.com/ceph/ceph/commit/20baf662112dd5f560bc3a2d2114b469444c3de8

A summary of the trace error is presented below:

- **Epoch number 2**. The monitor `mon1` is elected as the leader with the quorum of `mon1`, `mon2`, and `mon3`. The proposal number chosen in the collect phase is 102. The client requests the value `v1`. The monitor mon1 crashes before sending the begin messages to its peers. Some of the variable states before the next epoch:

  `monitor_store = {mon1: Nil, mon2: Nil, mon3: Nil};`

  `uncomitted values = {mon1: v1, mon2: Nil, mon3: Nil};`

- **Epoch number 4**. The monitor `mon2` is elected as the leader with the quorum of `mon2` and `mon3`. The proposal number chosen in the collect phase is 204. The client requests the value `v2`. The monitor `mon2` receives accept responses from `mon3` and commits the value `v2`. The monitor `mon1` is restored and elections are triggered before `mon3` receives the commit message with `v2`. Some of the variable states before the next epoch:

  `monitor_store = {mon1: Nil, mon2: v2, mon3: Nil};`

  `uncomitted values = {mon1: v1, mon2: Nil, mon3: v2};`

- **Epoch number 6**. The monitor `mon1` is elected as the leader with the quorum of `mon1`, `mon2`, and `mon3`. The monitor `mon2` crashes at the begin of the collect phase. Some of the variable states before the next epoch:

  `monitor_store = {mon1: Nil, mon2: v2, mon3: Nil};`

  `uncomitted values = {mon1: v1, mon2: Nil, mon3: v2};`

- **Epoch number 8**. The monitor `mon3` is elected as the leader with the quorum of `mon1` and `mon3`. The proposal number chosen in the collect phase is 308. In collect phase `mon3` must choose between `v1` or `v2` as the uncommitted value, when receiving the last message from the peer.

  This is the step that causes the error. In a normal scenario `mon3` would choose the value `v2` since it was the last one to be proposed. However, because of the introduced bug, `mon3` does not have information of when each of the values was proposed and ultimately decides on the value `v1`, violating the invariant. Some of the variable states before the next epoch:

  `monitor_store = {mon1: Nil, mon2: v2, mon3: Nil};`

  `uncomitted values = {mon1: v1, mon2: Nil, mon3: v2};`

This examples shows that the specification can be used to find real world errors and provides evidence that the specification matches the consensus algorithm that is implemented, since the bug introduced produces the same bug. Although this provides limited confidence, it is still better than specifying the algorithm in prose, since with a formal language there is tooling for helping verifying the safety of the algorithm.

For the second example, the specification, without the modifications, is checked using TLC. Fortunately, TLC yields no errors, which gives confidence that the implemented algorithm has no safety violations.

In Table 4.1 are some statistics from the model checker execution.

Both tests were executed with 20 workers on a machine with 2 sockets, each one a Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, and with 256GB of RAM.

Table 4.1: TLC execution statistics of the specification.

Execution	**Number of states (total / distinct)**	**Depth of state graph**	**Time elapsed**	**Status**
With bug	23245428 / 8927114	38	72971 ms (around 1 min)	Error found, invariant violated
Without bug	206551851 / 75709481	71	646171 ms (around 11 min)	No errors found

4.2 Performance of the Model

The performance analysis of a specification model is crucial to understand and find bottlenecks in the model checking step. This information can help improve the performance of the model, making it possible to check and verify a larger number of states.

This analysis is done by studying how the number of possible states changes depending on several factors. The factors studied in this section are the variables for limiting the search space (such as the number of epochs) and the structure of the system (such as the number of monitors).

4.2.1 Results

In this section the results obtained from the analysis of the model checking statistics are presented.

Figure 4.1 depicts the relation between the number of generated states and the maximum possible epoch (number of elections). The condition on the maximum epoch is used to limit the search space. Some important observations are that this condition has an exponential effect on the number of generated states and that the number of distinct states grows at a slower rate compared to the number of total states.

Figure 4.2 shows the relation between the number of generated states and the maximum proposal number. This condition on the proposal number is used to limit the search space. The distribution in the number of states is similar to the previous figure.

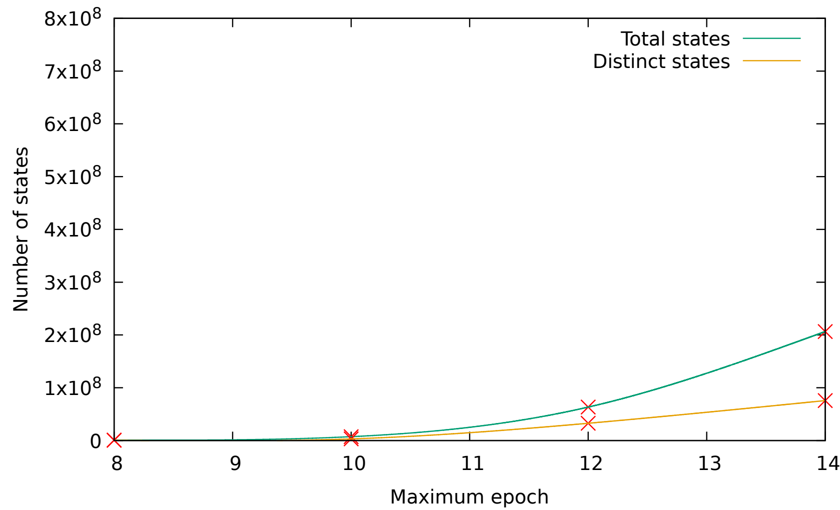


Figure 4.1: Number of states generated (total and distinct) depending on the maximum possible epoch. Model with 3 monitors, a set of 2 possible values, a proposal number limit of 400, and limit on the number of committed values of 2.

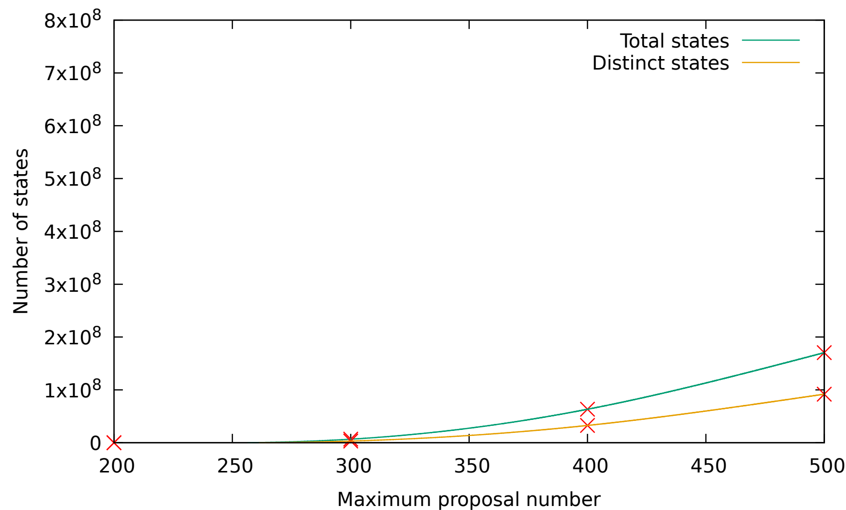


Figure 4.2: Number of states generated (total and distinct) depending on the maximum possible proposal number. Model with 3 monitors, a set of 2 possible values, a epoch limit of 12, and limit on the number of committed values of 2.

In the Figure 4.3 is presented the relation between the number of generated states and the maximum number of committed values. This condition on the number of committed values is used to limit the search space. This condition also has an exponential effect on the number of generated states and that the number of distinct states grows at a slower rate. Compared with the previous figures, this condition has greater impact in the number of states.

Figure 4.4 shows the relation between the number of generated states and the number of monitors. The number of monitors has an exponential effect on the number of generated states.

From all the factors presented, this has the biggest impact on the number of states generated. However, this factor is the less important because errors in the algorithm that are present with four or more monitors should also be present with three monitors. Therefore, the other three factors presented before are relatively more important to find errors in the algorithm.

The number of distinct states is lower that the number of total states because there are multiple ways to reach certain states (for example the non-determinism in the order that the nodes accept proposals).

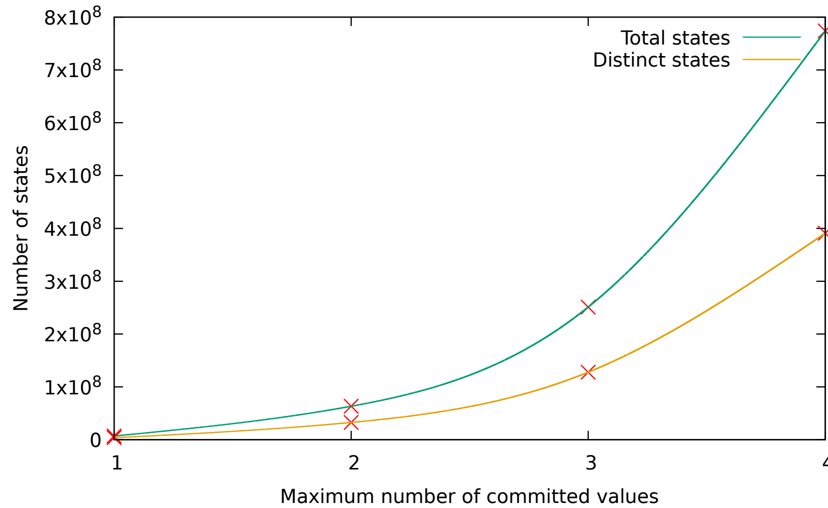


Figure 4.3: Number of states generated (total and distinct) depending on the maximum number of committed values. Model with 3 monitors, a set of 2 possible values, a epoch limit of 12, and a proposal number limit of 400.

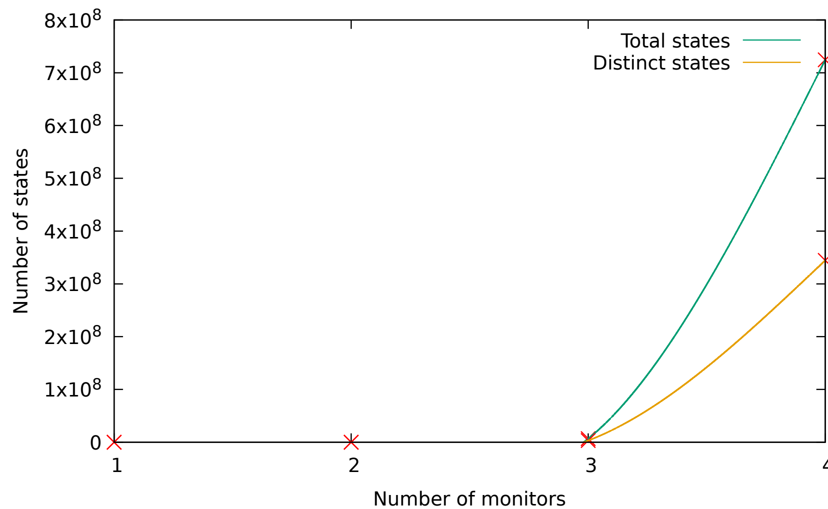


Figure 4.4: Number of states generated (total and distinct) depending on the number of monitors in the system. Model a set of 2 possible values, a epoch limit of 12, a proposal number limit of 300, and limit on the number of committed values of 2.

4.2.2 Summary

The graphics presented in this section help the user to understand what the bottlenecks are when scaling the model. Some important observations are:

- All factors have an exponential effect on the number of generated states.
- The number of distinct states grows at a slower rate than the number of total states.
- The factors with most impact are the number of monitors in the system and the limit on the number of committed values.

One of the reasons for the exponential growth of states in all the examples is the communication mechanism. For example, the message broadcast is nondeterministic and the model checker has to verify all orders in which the monitors receive a message.

The difference in distinct and total number of states comes from the generation of states that are equivalent to states previously explored (the model checker does not explore the repeated state again). One of the reasons for this is having multiple paths to a state, e.g. there are multiple ways to reach a state where a certain value has been chosen by all the nodes.

4.3 Performance of the Visualization Tool

One of the main features of graph explorer application previously presented is the ability to work with big graph files. In this section, the application is tested with state graphs of different sizes and the respective results are presented.

Figure 4.5 presents the relation between file sizes and the amount of memory that the application uses. To obtain these results, four state graphs were generated with varied sizes using the Ceph consensus algorithm specification with different search space limits. Then each file was loaded into an application instance on a Firefox browser and the memory used by the application was measured by taking a snapshot using the browser developer tools [2].

The previous graphic shows that the application reduces the amount of memory used to around ten per cent of the file size. This result is observed for all files tested in the application. By using less memory, the application is able to explore bigger state graphs, which can help the user to understand if the specification is modelling the algorithm or system correctly. This can also help a user improve the performance of the specification by helping identify possible repeated states that are being explored.

[2] https://developer.mozilla.org/docs/Tools/Memory

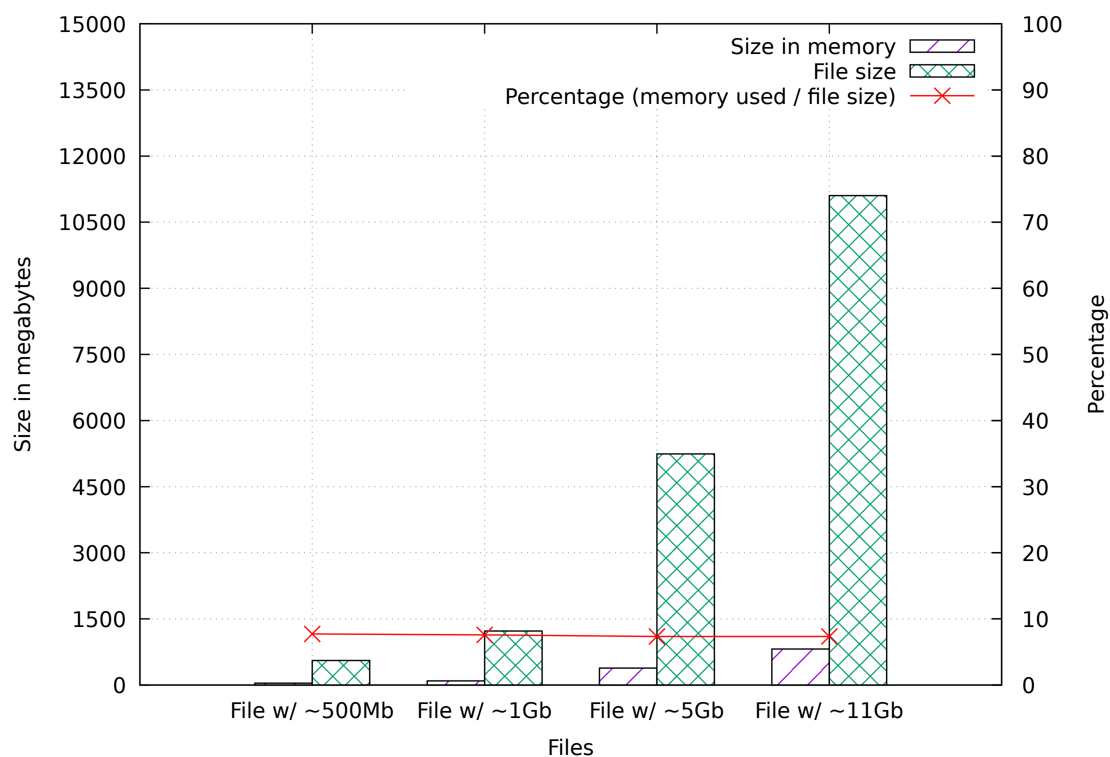


Figure 4.5: Memory used by the visualization tool depending on the file size. Results obtained using the Firefox browser (Version 89.0.1) for Ubuntu 20.04.

Chapter 5

Conclusion

5.1 Research Summary

This thesis explores the use of formal methods to test properties and check correctness of a real world system. The object of this work was the consensus algorithm used in Ceph, which was formalized and tested in TLA$^+$.

The specification was validated by introducing a real world bug present in an early version of Ceph. This bug produced the same error both in the implementation and in the specification, providing evidence that the specification matches the algorithm that is implemented in Ceph. Even though the evidence provided is limited, it is better than specifying the algorithm in prose, because in a TLA$^+$ it can be simulated and verified by a computer.

The specification (without the bug) was also evaluated with the model checker and no errors were found, which gives confidence that there are no design errors in the current version of the consensus algorithm implemented in Ceph. The tool developed for visualization helped in improving the efficiency of the model, in understanding how the algorithm works, and in understanding error traces produced by the model checker.

The results obtained by testing the specification showed that the number of generated states has an exponential growth. However, the search space does not have to be large for the model checker be able to find interesting and important bugs, such as the one presented before. In the results obtained for the visualization tool, the memory usage of the tool was efficient which enables its usage for bigger test cases that can result in better insights.

5.2 Future Work

The work presented in this thesis is easily available on GitHub[1] to be used in future projects.

The specification presented can be further explored in order to test new properties. For example, temporal properties would be an interesting subject to research the liveness of the algorithm. The specification can also be modified to evaluate new versions of algorithms for consensus to be used in Ceph, such as an approach that uses Byzantine Fault-Tolerance (BFT).

5.3 Conclusions

The paradigm of distributed systems is being increasingly used, both for scaling requirements and for failure tolerance (to ensure minimal downtime). However, these systems are normally complex and prone to failures, which makes it challenging to designing ant implement applications in a distributed setting. The algorithms for synchronization between the nodes in a distributed system are also long and complex, and therefore hard to implement.

In this work, the use of formal methods was explored with the intent of helping the implementation and verification of distributed algorithms. For this goal, the algorithm used for consensus and synchronization in Ceph was specified in the TLA$^+$ formal language. The specification was helpful in verifying the correctness of the algorithm and finding complex and critical bugs. The formalization of the algorithm also helped to understand it and explore execution flows, with the help of the visualization tool developed in this work.

In conclusion, the area of formal methods is growing and proving to be successful in the design and in the analysis of complex algorithms. Many companies in the industry are adopting formal methods as a way to ensure safety of critical parts of their products. The use of formal methods should grow as the development of complex application in critical scenarios increases and with the appearance of more formal languages and tools that are more accessible, as it is the case with TLA$^+$.

[1] https://github.com/afonsonf/ceph-consensus-spec and https://github.com/afonsonf/tlaplus-graph-explorer

Bibliography

[1] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Consensus in the Cloud: Paxos Systems Demystified. In *25th International Conference on Computer Communication and Networks, ICCCN 2016, Waikoloa, HI, USA, August 1-4, 2016*, pages 1–10. IEEE, 2016. doi:10.1109/ICCCN.2016.7568499.

[2] J. Aspnes. Time- and Space-Efficient Randomized Consensus. *Journal of Algorithms*, 14(3): 414 – 431, 1993. ISSN: 0196-6774. doi:https://doi.org/10.1006/jagm.1993.1022.

[3] Brannon Batson and Leslie Lamport. High-Level Specifications: Lessons from Industry. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 242–261. Springer, 2002. doi:10.1007/978-3-540-39656-7_10.

[4] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[5] Alysson Neves Bessani and Eduardo Alchieri. Chapter 4 A Guided Tour on the Theory and Practice of State Machine Replication. 2014.

[6] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:8, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN: 978-3-95977-169-6. doi:10.4230/OASIcs.FMBC.2020.10.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[8] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. Formal Verification of Multi-Paxos for Distributed Consensus. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi,

and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 119–136, 2016. doi:10.1007/978-3-319-48989-6_8.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[10] Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987. doi:10.1145/7531.7533.

[11] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal ACM*, 32(2):374–382, April 1985. ISSN: 0004-5411. doi:10.1145/3149.214121.

[13] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. ISSN: 0163-5700. doi:10.1145/564585.564601.

[14] Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, USA, 1994.

[15] Martin Hilbert and Priscila López. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65, 2011. ISSN: 0036-8075. doi:10.1126/science.1200970.

[16] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient byzantine fault tolerance. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 295–308. ACM, 2012. doi:10.1145/2168836.2168866.

[17] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019. doi:10.1145/3360549.

[18] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978. doi:10.1016/0376-5075(78)90045-4.

[19] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

[20] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3): 872–923, May 1994. ISSN: 0164-0925. doi:10.1145/177492.177726.

[21] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

[22] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[23] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002. ISBN: 032114306X.

[24] Leslie Lamport. Fast Paxos. *Distributed Comput.*, 19(2):79–103, 2006. doi:10.1007/s00446-006-0005-x.

[25] Leslie Lamport. Byzantizing Paxos by Refinement. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011. doi:10.1007/978-3-642-24100-0_22.

[26] Leslie Lamport and Mike Massa. Cheap Paxos. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 307–314. IEEE Computer Society, 2004. doi:10.1109/DSN.2004.1311900.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[28] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN: 978-3-642-17511-4.

[29] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Formal Verification Of Pastry Using TLA+. In Leslie Lamport and Stephan Merz, editors, *International Workshop on the TLA+ Method and Tools*, Paris, France, August 2012.

[30] Rolando Martins, Manuel E. Correia, Luís Antunes, and Fernando Silva. Iris: Secure reliable live-streaming with opportunistic mobile edge cloud offloading. *Future Generation Computer Systems*, 101:272 – 292, 2019. ISSN: 0167-739X. doi:https://doi.org/10.1016/j.future.2019.06.011.

[31] Microsoft. Azure Cosmos TLA+ specifications. https://github.com/Azure/azure-cosmos-tla, 2018. Accessed: 2021-05-12.

[32] Chris Newcombe. Why Amazon Chose TLA +. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of

Lecture Notes in Computer Science, pages 25–39. Springer, 2014. doi:10.1007/978-3-662-43652-3_3.

[33] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery. ISBN: 0897912772. doi:10.1145/62546.62549.

[34] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[35] Joao Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 51–58. IEEE Computer Society, 2018. doi:10.1109/DSN.2018.00018.

[36] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3.

[37] Daniel van der Ster and Arne Wiebalck. Building an organic block storage service at CERN with Ceph. *Journal of Physics: Conference Series*, 513(4):042047, jun 2014. doi:10.1088/1742-6596/513/4/042047.

[38] Eric Verhulst, Gjalt G. de Jong, and Vitaliy Mezhuyev. An Industrial Case: Pitfalls and Benefits of Applying Formal Methods to the Development of a Network-Centric RTOS. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 411–418. Springer, 2008. doi:10.1007/978-3-540-68237-0_29.

[39] Paulo Verissimo and Luis Rodrigues. *Distributed systems for system architects*, volume 1. Springer Science & Business Media, 2012.

[40] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221.

[41] Sage A. Weil. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, University of California, Santa Cruz, 2007.

[42] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006. doi:10.1109/SC.2006.19.

[43] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999. doi:10.1007/3-540-48153-2_6.

Appendix A

Specification Configuration

```
CONSTANTS
m1 = m1    m2 = m2    m3  = m3
v1 = v1    v2 = v2    Nil = Nil

CONSTANTS
Monitors = {m1,m2,m3}    Value_set = {v1, v2}

CONSTANTS
OP_COLLECT = OP_COLLECT   OP_LAST      = OP_LAST
OP_BEGIN   = OP_BEGIN     OP_ACCEPT    = OP_ACCEPT      OP_COMMIT  = OP_COMMIT
OP_LEASE   = OP_LEASE     OP_LEASE_ACK = OP_LEASE_ACK

CONSTANTS
STATE_RECOVERING        = STATE_RECOVERING
STATE_ACTIVE            = STATE_ACTIVE
STATE_UPDATING          = STATE_UPDATING
STATE_UPDATING_PREVIOUS = STATE_UPDATING_PREVIOUS
STATE_WRITING           = STATE_WRITING
STATE_WRITING_PREVIOUS  = STATE_WRITING_PREVIOUS
STATE_REFRESH           = STATE_REFRESH
STATE_SHUTDOWN          = STATE_SHUTDOWN

CONSTANTS
PHASE_ELECTION     = PHASE_ELECTION
PHASE_SEND_COLLECT = PHASE_SEND_COLLECT    PHASE_COLLECT    = PHASE_COLLECT
PHASE_LEASE        = PHASE_LEASE           PHASE_LEASE_DONE = PHASE_LEASE_DONE
PHASE_BEGIN        = PHASE_BEGIN           PHASE_COMMIT     = PHASE_COMMIT

SYMMETRY SYMM
CHECK_DEADLOCK FALSE
INVARIANT Inv
INIT Init
NEXT Next
VIEW view
```