Uma Implementação Fiel do Algoritmo Generalized Paxos e uma *CStruct* para o Problema de Coordenação de Lease Distribuído

Tuanir França Rezende

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2017

Tuanir França Rezende

Uma Implementação Fiel do Algoritmo Generalized Paxos e uma *CStruct* para o Problema de Coordenação de Lease Distribuído

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Lásaro Jonas Camargos

Uberlândia
2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

R467i
2017

Rezende, Tuanir França, 1992-
Uma implementação fiel do algoritmo Generalized Paxos e uma CStruct para o problema de coordenação de Lease Distribuído / Tuanir França Rezende. - 2017.
87 f.

Orientador: Lásaro Jonas Camargos.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Disponível em: http://dx.doi.org/10.14393/ufu.di.2018.141
Inclui bibliografia.

1. Computação - Teses. 2. Algoritmos - Teses. 3. Análise de sistemas (Computação) - Teses. 4. Otimização estrutural - Teses. I. Camargos, Lásaro Jonas. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Rejâne Maria da Silva – CRB6/1925

Este trabalho é dedicado a minha mãe Nilcéa e meu pai Tuanir.

Agradecimentos

Gostaria de agradecer a todos que ajudaram direta ou indiretamente na realização deste trabalho. Especialmente ao meu orientador Lásaro pela confiança e dedicação e a meu amigo Rodrigo Queiroz Saramago.

Agradeço também a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro, indispensável.

"Passion has little to do with euphoria and everything to do with patience. It is not about feeling good. It is about endurance. Like patience, passion comes from the same Latin root: pati. It does not mean to flow with exuberance. It means to suffer."

(Mark Z. Danielewski, House of Leaves)

Resumo

Replicação de Máquina de Estados é uma abordagem amplamente aceita para se construir um sistema distribuído tolerante a falhas. Nesta técnica, as réplicas entram em acordo a respeito da sequência de comandos que irão executar, o que consiste no problema fundamental do consenso distribuído. Devido a definição básica do problema do consenso, as soluções existentes mais conhecidas para o mesmo não relevam o fato de que não é necessária a definição de uma ordem total para a sequência de comandos que as réplicas executam. O algoritmo chamado: *Generalized Paxos* resolve uma versão mais genérica e versátil do problema do consenso, que exige apenas que as réplicas entrem em acordo a respeito de uma ordem parcial da sequência de comandos. Além de ser uma das versões mais otimizadas para se implementar replicação de máquina de estados, o algoritmo *Generalized Paxos* é capaz de resolver diferentes tipos de problema de acordo. Apesar disso, o potencial do algoritmo não é totalmente explorado devido a sua alta complexidade e a baixa quantidade de estudos e implementações do mesmo. Este trabalho tem o intuito de diminuir as lacunas existentes entre a teoria e prática no *Generalized Paxos*, através da implementação do algoritmo e otimizações que podem ser aplicadas à mesma. Além disso, atestando em favor do uso de *Generalized Paxos* em problemas do mundo real, este trabalho também fornece uma nova formalização que permite que o algoritmo solucione uma variação do problema de coordenação de *leases* em ambientes distribuídos.

Palavras-chave: Consenso Distribuído, Consenso Generalizado, Paxos, Generalized Paxos, Sistemas Distribuídos, Replicação de máquinas de estado, Coordenação de *leases*.

A Faithful Generalized Paxos Implementation and a Novel *CStruct* for Distributed Lease Coordination

Tuanir França Rezende

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2017

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO – FACOM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO – PPGCO

The undersigned hereby certify they have read and recommend to the PPGCO for acceptance the dissertation entitled **"A Faithful Generalized Paxos Implementation and a Novel CStruct for Distributed Lease Coordination"** submitted by **"Tuanir França Rezende"** as part of the requirements for obtaining the **Master's degree in Computer Science**.

Uberlândia, 5 de Janeiro de 2017

Supervisor: _____

Prof. Dr. Lásaro Jonas Camargos
Universidade Federal de Uberlândia

Examining Committee Members:

_____

Prof. Dr. Luis Fernando Faina
Universidade Federal de Uberlândia

_____

Prof. Dr. Gustavo Maciel Dias Vieira
Universidade Federal de São Carlos

Abstract

State Machine Replication is a broadly accepted approach to build a fault-tolerant distributed system. In this technique, replicas agree on a sequence of commands that they will execute. This consists in the fundamental Consensus problem in distributed systems. The classical Consensus problem and its solutions do not leverage the fact that the sequence of commands executed by the replicas does not need to follow a total order. The algorithm entitled Generalized Paxos solves a more generic and versatile version of the Consensus problem by requiring that the replicas agree on a partially ordered sequence of commands. As well as being one of the most optimized solutions to implement state machine replication, Generalized Paxos is also capable of solving different agreement problems. However, due to its high complexity and the small amount of research attesting its benefits, the algorithm's potential is not fully explored. This work aims at diminishing the gap between theory and practice regarding Generalized Paxos, through an implementation of the algorithm and optimizations that can be applied to it. Furthermore, to advocate the use of Generalized Paxos in real-world problems, this work provides a new construct that allows the algorithm to solve a variation of the lease coordination problem in distributed systems.

Keywords: Distributed Consensus, Generalized Consensus, Paxos, Generalized Paxos, Distributed Systems, State Machine Replication, Lease Coordination.

List of Figures

Acronyms list

CFABCAST Collision-fast Atomic Broadcast

CAPES Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

DAG Directed Acyclic Graph

FGGC Fast Genuine Generalized Consensus

GDS Geographically Distributed Systems

JVM Java Virtual Machine

LOC Lines of Code

LAN Local Area Network

POB Partial Order Broadcast protocol

RSM Replicated State Machine

TLA+ Specification Language for Temporal Logic of Actions

WAN Wide Area Network

List of Algorithms

Contents

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

student name and signature

Chapter 1

Introduction

Distributed computing has become a ubiquitous model of computing in our lives due to the rise, success and necessity of fault-tolerant services. This kind of services requires data redundancy and thus, the necessity of keeping these data consistent. For this purpose, the Replicated State Machine (RSM) technique is commonly used, which consists in replicating identical process through multiple nodes and coordinating the interaction of clients with these replicas (LAMPORT, 1978).

Classic Paxos is a cornerstone distributed computing algorithm (LAMPORT, 1998), that is used to provide a fault-tolerant implementation of a state machine in an asynchronous message-passing system (e.g. the Internet). Nowadays, Paxos is used as a fundamental building block in many fault-tolerant real-world systems that require high throughput and low latency. Therefore, a lot of work has been made in order to find solutions to some of its weaknesses. One of the most interesting efforts to improve Paxos is called: Generalized Paxos (GPaxos) which is the algorithm explored in-depth in this work.

1.1 Motivation

Consensus is a central primitive for building replicated systems, but its latency constitutes a bottleneck. In a recent paper, the well-known solution to consensus: Fast Paxos (LAMPORT, 2006a) is enhanced by leveraging the commutativity of concurrent commands. This new primitive, called Generalized Paxos (GPaxos) (LAMPORT, 2004) reduces the collision rate, and thus the latency of Fast Paxos by solving a problem more general than consensus, entitled: Generalized Consensus.

Generalized Consensus is a distributed problem to which several key agreement problems reduce, and by taking advantage of this theoretical framework we envision that GPaxos may unify within a single and novel Agreement-as-a-Service infrastructure to multiple distributed protocols. To date, this potential is however not fully unleashed, due to the steep learning curve of the protocol and the high complexity of its implementation.

Moreover, before GPaxos reaches a real-world usage, several computationally expensive operations must be optimized and simplified.

1.2 Thesis Organization

The rest of this thesis is organized in the following manner. In Chapter 2 we present all the theory on which this work is based on, providing brief explanations to the state-of-the-art works in distributed computing that relates to the themes in this thesis. Chapter 3 describes the architecture of the implementation and the modules used from the framework, as well as details of the implementations of the protocol abstractions. Chapter 4 presents our alternative way to fast detect an agreement in one of the final phases of the algorithm, as well as our new solution to a version of the distributed lease coordination problem utilizing Generalized Paxos instantiated as a new *CStruct* set we defined. Finally, Chapter 5 concludes the work through a discussion of the possible future work that, among other topics, focuses on what would have to be done in order to achieve high performance in our implementation.

1.3 Research Contributions

This work is one of the few efforts for closing the gap between theory and practice regarding GPaxos. To this end, we first provided a complete explanation of the protocol, hardly found elsewhere, together with all the theoretical background necessary to understand the Generalized Consensus problem. The protocol investigation resulted in a modern and faithful open source implementation of GPaxos[1] that presents an unprecedented level of extensibility acquired through the use of a new framework for the study of distributed consensus.

To assess the versatility of the Generalized Consensus problem, we presented a new *CStruct* set that can be applied to GPaxos that solves a variation of the distributed lease coordination problem. Our last contribution consists in optimizations that apply to the critical phases of the algorithm: *(i)* a method to quickly start a new round, *(ii)* an algorithm to the detection of an agreement with a considerately better order of complexity than the naive approach.

Parts of the results were submitted as a paper entitled *On Making Generalized Paxos Practical* to *The 31st IEEE International Conference on Advanced Information Networking and Applications* in a collaboration with Pierre Sutra, the creator of Fast Genuine Generalized Consensus (FGGC) (SUTRA; SHAPIRO, 2011), one of the few Generalized Paxos specialists in the world.

[1] <https://bitbucket.org/tfr/gpaxossimple>

Chapter 2

Fundamentals

In this chapter we present the theoretical background necessary to fully understand the building blocks that compose Generalized Paxos.

2.1 System Model

We assume a system model with an infinite number of processes, which may fail by crashing, but do not behave maliciously. If a process does not fail, it is *correct*, and otherwise, it is *faulty*.

For a finite subset of processes, which play an essential role in the algorithms, as we discuss later, the crash-recovery model is assumed, as their state must be recovered to ensure correctness. Finally, even if the algorithms are crash-recovery, our implementation abides by the fail-stop (crash) model (BERNSTEIN; HADZILACOS; GOODMAN, 1987).

Regarding synchrony, we assume a model similar to the timed asynchronous model defined in (CRISTIAN; FETZER, 1999), in which each process makes progress at its own speed. That is, processes have access to local clocks, which are not synchronized with each other, and there are no bounds to the time needed to execute actions.

Last, communication happens through message passing over reliable asynchronous channels, which may delay the messages for a bounded (yet unknown) amount of time. Messages are neither duplicated nor have their content altered.

2.2 Distributed Consensus

Considering a set of processes that can fail by crashing, Consensus is a fundamental problem of distributed systems traditionally defined as a procedure to agree on one out of a set of input values from each of these process. In this work, however, we use an alternative definition in which only a subset of processes can input values, and not even all of them must have an input. This version was proposed by Lamport and separates the concerns involved in reaching agreement in a way that simplifies the protocols. In

(LAMPORT, 2004), Lamport defines the consensus problem in terms of a set of *proposer* processes that propose values and a set of *learner* processes that must agree upon a value. Think of the proposers as a system's clients and the learners as the servers that cooperate to implement the system.

The resulting consensus problem has the following three safety requirements:

Nontriviality Any value learned must have been proposed.

Stability A learner can learn at most one value. (In other words, it cannot change its mind about what value it has learned.)

Consistency Two different learners cannot learn different values.

The safety properties are required to hold under certain failure assumptions. For asynchronous systems, they are generally required to hold despite any number of non-Byzantine failures (i.e. processes cannot act in maliciously and arbitrary manners, for example: incorrectly processing messages and/or forging its identity).

The consensus problem also has the following liveness requirement:

Liveness(C, l) If value C has been proposed, then eventually learner l will learn some value.

Liveness is usually stated in terms of C and l because the assumption under which it must hold generally depends on these parameters. For asynchronous implementations, the usual assumption is that learner l, the proposer of C, and a sufficient number of other processes are nonfaulty and can communicate with one another.

2.2.1 On Acceptors and Quorums

Consensus is implemented using a finite set of processes called *acceptors*. Acceptors, proposers, learners are processes and one can often find in the literature the term *agent* used to describe each of this processes. Each process is executed on a node, and a single node may execute several of these processes. A set of nodes is considered to be nonfaulty iff the set of processes executed by those nodes is nonfaulty, a process is nonfaulty iff it never fails.

When we have a sufficiently large number of acceptors that guarantee liveness, we call it a *quorum Q*. *Q* is a *quorum* iff condition *Liveness(C, l)* of consensus holds when the set S is nonfaulty for a long enough period of time, where S is the set that must contain the process that proposed command C, the learner l and the acceptors in Q. This condition is required to hold regardless of what may have happened before S became nonfaulty.

Proposer, acceptor, and learner can be viewed as roles performed by the nodes in a distributed system. The relation between these roles and the roles of client and server can vary from system to system. Typically, the clients are proposers and the server nodes are

both learners and acceptors. The clients may also be considered to be learners. What roles a node plays is an implementation choice that determines the fault-tolerance properties of the system (LAMPORT, 2004).

2.2.2 A Note On Synchronous and Asynchronous Distributed Systems

Consider a synchronous distributed system in which each pair of processes is connected through a channel and the only possible failure is the crash of processes. To make things even simpler, we assume that only communication takes time and that when a process receives an *inquiry message* it responds to it right away (i.e in zero time). Let Δ be the upper bound of the round-trip communication delay. In this scenario, a process P_i can determine with ease the crash of another process P_j. Process P_i sets a timer's time-out period to Δ and sends an *inquiry message* to P_j. If P_i receives an answer before the expiration of the timer, it can safely conclude that P_j has not crashed before receiving the *inquiry message* (this concept is illustrated in Figure 1).

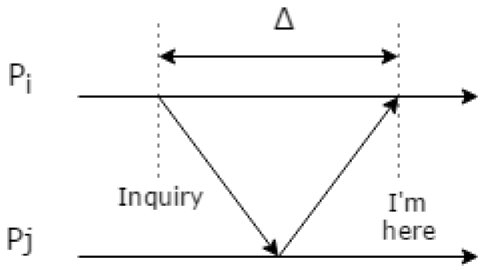

Figure 1 – Communication is timely.

If P_i has not received an answer from P_j when the timer expires, it can safely conclude that P_j has crashed (see Figure 2).

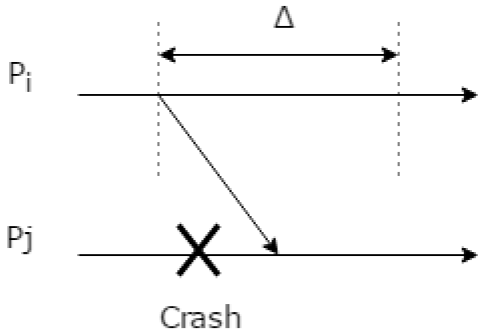

Figure 2 – Process P_j has crashed.

Considering asynchronous systems, its processes can use timers but cannot fully trust them, irrespective of the time-out period. This is due to the fact that message transfer

delays in distributed systems have no upper bound. Note that if P_i uses a timer (with some time-out period Δ) to try to detect the crash of P_j, the two scenarios presented in Figures 1 and 2, can occur. Furthermore, a third scenario can also exist (Figure 3), the timer Δ can expire while the response is still on its way to P_i. This occurs because Δ is not the correct upper bound for a round-trip delay.

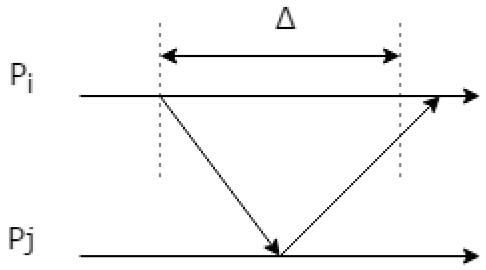

Figure 3 – The link between P_i and P_j is slow.

The fundamental difference between a synchronous and an asynchronous distributed system is that only the first and second scenarios can occur in a synchronous distributed system, while all three scenarios can occur in an asynchronous distributed system (RAYNAL; SINGHAL, 2001). In such a system, P_j cannot distinguish the second and third scenarios when its timer expires. It is important that one understand the concepts presented in this section for they are closely related to the fundamental FLP result in distributed systems that is explained in the following section.

2.2.3 On the FLP result

It is important to understand the difficulty behind reaching consensus in an asynchronous environment. When the only possible failures are process crashes, the consensus problem has relatively simple solutions in synchronous distributed systems, but this is not the case in asynchronous distributed systems.

The FLP result (FISCHER; LYNCH; PATERSON, 1985) states that it is impossible to design a deterministic protocol solving the consensus problem in an asynchronous system if there exists the possibility of even a single process failure. Intuitively, this is due to the impossibility of safely distinguishing a very slow process from a crashed process in an asynchronous context (remember the three scenarios of an asynchronous system in section 2.2.2). This impossibility result has motivated researchers to discover a set of minimal properties that, when satisfied by an asynchronous distributed system, make the consensus problem solvable.

Hopefully, real systems satisfy partial-synchrony assumptions which are sufficient to solve consensus. For instance, in the model of unreliable failure detectors, it has been shown that the eventual leader oracle is both sufficient and necessary to solve consensus (CHANDRA; TOUEG, 1996).

2.2.4 Lower-Bound Results for Consensus

Before presenting the algorithms: Paxos and Fast Paxos, which are solutions to the distributed consensus problem and Generalized Paxos, a solution to the generalized consensus problem, it helps to know what is theoretically possible.

We proceed to describe some lower-bound results for consensus in asynchronous systems. The proof for each theorem can be found in (LAMPORT, 2004). We present each theorem along with notes from the author of (LAMPORT, 2004) to help make the implications of the results clearer.

Theorem 1 *Any two quorums have non-empty intersection.*

A consensus algorithm using N acceptors is said to tolerate F faults if every set of $N-F$ acceptors is a quorum. Theorem 1 implies that this is only possible if $N > 2F$.

Theorem 2 *Learning is impossible in fewer than 2 message delays.*

We define a set Q of acceptors to be a fast quorum iff, for every proposer p and learner l, there is an execution involving only the set of processes $Q \cup \{p, l\}$ in which a value proposed by p is learned by l in two message delays. Events that do not influence l's learning event are irrelevant, and processes could fail before they are performed. We can, therefore, assume that the communication in such an execution consists only of p proposing a value and sending messages to the acceptors in Q and to l, followed by the acceptors in Q sending messages to l.

Theorem 3 *If Q_1 and Q_2 are fast quorums and Q is a quorum, then $Q_1 \cap Q_2 \cap Q$ is non-empty.*

Comparing Theorem 3 with Theorem 1 allows one to infer that fast quorums have to be bigger than plain quorums.

We say that an algorithm with N acceptors is fast learning despite E faults iff every set of $N-E$ acceptors is a fast quorum. Theorem 3 implies that such an algorithm that tolerates F faults is possible only if $N > 2E + F$.

Theorem 4 *If for every acceptor a, there is a quorum not containing a, then a consensus algorithm cannot ensure that, even in the absence of failures, every learner learns a value in two message delays.*

Theorem 4 shows that it is impossible for a fault-tolerant consensus algorithm to guarantee, even in the absence of failure, that a value is always learned in two message delays. Its proof shows that this can't occur because two proposers can concurrently propose different values.

2.3 Paxos

Paxos (LAMPORT, 1998; LAMPORT, 2001) is a largely studied leader based consensus protocol and the most commonly used in the industry, for example in the *Chubby Lock Service* from Google (CHANDRA; GRIESEMER; REDSTONE, 2007).

Paxos agents (the same used to define the consensus problem in 2.2) execute multiple ballots (i.e. rounds), which are totally ordered by a relation $<$. Each ballot has a number associated with it, typically represented as a natural number, and each round is coordinated by one specific proposer. Even though there is a total order among ballot numbers, the ballot execution doesn't need to follow this order, and actions in different ballots may even interleave.

In Paxos, clients issue commands (also known as values) wrapped in proposal messages to proposers agents, but only a proposer that *believes itself to be the leader of the consensus instance* (or coordinator) has the rights to propose a command to the set of acceptors. To propose, the coordinator must start one of its ballots, bigger than any other previously started. Its proposal will become the *chosen* value of the instance, iff a correct and reachable quorum of acceptors for that ballot exists and no other process has the wrong idea of being the coordinator, which could lead multiple ballots being started, preventing each other from terminating.

A ballot in Paxos is divided into two phases: the first phase, referenced in the literature as *configuration* phase or *prepare* phase, serves the purpose of identifying previously chosen values and the second phase, referenced as the *accept* phase, tries to get some value chosen in some ballot. The first phase is comprised of actions: *Phase1a, Phase1b* and the second one of actions: *Phase2a, Phase2b*. One must understand that actions *Phase1a, Phase2a* initiate each phase, while actions *Phase1b, Phase2b* may be seen as responses to the previous actions. The set of actions can be seen in Figure 4 as well as the normal flow of messages (considering no failures).

As we mentioned before, one of the proposers has the role of the coordinator and is responsible for starting each phase of the ballot, by executing *Phase1a* and *Phase1b*. The actions *Propose* and *Learn* complete the algorithm, the former is executed by the proposers to propose a value and the later is executed by the learners to learn the decision of a consensus instance.

A learner l can only learn the decision of an instance after the execution of the two phases of some ballot m. Furthermore, a learner l can only learn a value that has been proposed by some proposer p. It's important to note that the *Propose* action must happen before the second phase, but not necessarily before the first one.

The algorithm works in the following way, the *prepare* phase is initiated by the leader through the action *Phase1a*, that consists in sending a $\mathsf{1A}$ message to the acceptors, when the acceptors receive this message, they do the following:

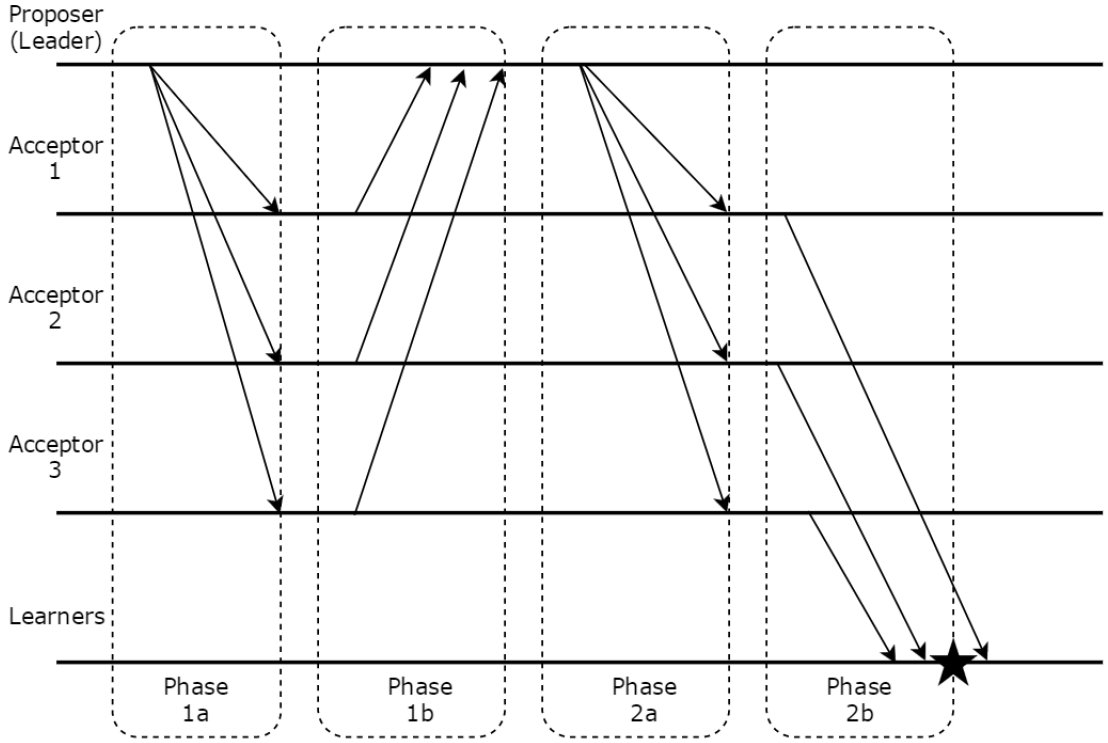

Figure 4 – Classic Paxos message exchange and phases in the lack of failures. The star marks the learning.

- If the acceptor has not responded to any 1A message, he updates its ballot to the ballot of the message 1A (we will call it ballot i) and sends a confirmation 1B message to the leader, through the execution of action *Phase1b*. This way it is guaranteed that these acceptors will not accept values for ballots smaller than i.

- If the acceptor already responded to some 1A message in some ballot s smaller to the current one (i), two scenarios are possible:

  - The acceptor has not yet received any 2A message with a proposal, sent during the coordinator's *accept* phase. In this case, the acceptor updates its ballot to the greatest ballot received (i) and sends a 1B message with this ballot i as parameter to the leader;

  - The acceptor has received some 2A message in some ballot k and it must have received a value v proposed by some proposer acting as coordinator in this ballot k ($k \leq s < i$), this means that v is the accepted value (its "vote") by this acceptor in the last ballot k in which he accepted (voted for) something. The acceptor then sends back a 1B message to the coordinator of i, containing as parameters (i, k, v), which are respectively the current ballot, the latest ballot k in which the acceptor accepted something and the value accepted in k.

The second phase *accept* is initiated once the coordinator receives messages for the round i from all acceptors in a quorum Q. The coordinator then executes action *Phase2a* sending a 2A message to the acceptors, the message contains as parameter (i, v), which are respectively the current round and the proposal value v selected by the coordinator. The picking of the value is based on these criteria:

- If the coordinator has received one or more 1B messages with values accepted by the acceptors, he then selects the value v of the proposal with the highest ballot number.
- If no 1B message received has any accepted value, the coordinator picks any value to be the proposal.

When an acceptor receives a 2A message with a value v for the round i, he accepts the proposal (i.e. votes for it) if it has not responded to a ballot larger than i. Then, the acceptor through the execution of action *phase2b* sends a 2B message confirming the accepted value v to the set of learner agents. Its important that one understands the distinction between an *accepted* value and a *decided* (or *chosen*) value; the former only means that the acceptor "voted" for such value, while the latter means that a quorum of acceptors voted for the same value in the same round and that the value will eventually be learned by the learners.

Finally, if a learner l receives 2B messages from a quorum of acceptors containing a value v, then it knows that v was chosen and can be safely learned.

In case of failure suspicion, the coordinator initiates a new ballot by executing the *prepare* phase again. Since a coordinator sends the value to be accepted only at the beginning of the second phase (i.e. *accept* phase), the first phase of the algorithm can be executed before receiving any proposal. On a real application, probably many consensus instances will be needed, and the coordinator can execute the *prepare* phase *a priori* for all consensus instances. Thus, the amortized latency for solving each instance becomes only three messages steps if there are no failures and no other coordinator interferes by starting a higher-numbered round: one step for the proposal to reach the leader and two more for the second phase of the leader's current ballot.

2.4 Fast Paxos

Fast Paxos (LAMPORT, 2006b), is an extension of Paxos algorithm in which proposers other than the coordinator propose directly to the acceptors, therefore, reducing the latency of reaching a decision in one communication step in good scenarios for all proposers.

The main difference that allows Fast Paxos to have this reduced latency is that Fast Paxos has two types of ballots, *fast* and *classic*. The *classic* ballots are similar to Paxos,

while the *fast* ballots require larger quorums in the first phase and accept proposals from all proposers in the second phase.

Although the existence of fast ballots may reduce the latency necessary to a value to be learned, there is a trade-off in the form of a requirement of bigger quorum size during this type of ballots, as seen in 2.2.4. For example, if classic quorums are defined to be any majority of acceptors, fast quorums must be as big as $\lceil 3n/4 \rceil$ acceptors.

The existence of fast ballots also introduces the possibility of something called a *collision* (LAMPORT, 2006a). Since any proposer can send its proposals directly to the set of acceptors, different acceptors may accept different values in the same round, possibly leading to a situation where no value is decided at the end of a ballot. To recover from such a situation, a new ballot must be started.

In an effort to circumvent the problem of collisions, a series of new algorithms were devised. One of them is the focus of this work, Generalized Paxos, a natural extension of Fast Paxos that tackles the collision problem by solving a problem more generic than consensus, called: Generalized Consensus. We proceed by describing this problem in-depth and in the sequence explaining Generalized Paxos.

2.5 Generalized Consensus

Both Paxos and Fast Paxos were created to solve the distributed consensus problem, so their inner workings are tied to this problem definition. The working of Generalized Paxos cannot be fully understood without a precise definition of the problem it solves: Generalized Consensus, so with this section, we provide a complete study of it.

Generalized Consensus is an extension of the ordinary Consensus problem, defined in terms of a data structure called *command structure*, or simply *c-struct*. Depending on the c-structs in use, Generalized Consensus reduces to various problems, such as Consensus, Total Order Broadcast or Generic Broadcast (PEDONE; SCHIPER, 2002). In what follows, we present the Generalized Consensus problem and the notion of c-struct using the original framework of (LAMPORT, 2004), but first, we present some standard notation and simple mathematics that will be used throughout this work.

2.5.1 Mathematical Preliminaries

Here we present mathematical background borrowed from (LAMPORT, 2004).

2.5.1.1 Notation

We use the customary operators of propositional logic and predicate calculus, with $\forall x \in S : P$ and $\exists x \in S : P$ asserting that P holds for all and for some x in s, respectively,

and we let $\stackrel{\triangle}{=}$ mean *is defined to equal*. We use the following notation for representing sets:

- $\{e_1, \ldots, e_n\}$ is the set consisting only of the elements e_i and $\{\}$ is the empty set.
- $\{x \mid P\}$ is the set of all x such that P is true.

2.5.1.2 Equivalence Classes

A relation $\sim$ on a set S is called an *equivalence relation* iff it satisfies the following properties, for all u, v , and w in S :

- Reflexive: $u \sim u$;
- Comutative: $u \sim v$ iff $v \sim u$; and,
- Transitive: $u \sim v$ and $v \sim w$ imply $u \sim w$.

We define $[u]$, the equivalence class under $\sim$ of an element u in S, as

$$[u] \stackrel{\triangle}{=} \{v \in S \mid v \sim u\}$$

Thus, $u \sim v$ iff $[u] = [v]$. The set of all such equivalence classes is called the *quotient space* of S *under* $\sim$, and is written $S/\sim$

2.5.1.3 Directed Graph

A directed graph consists of a pair $\langle N, E\rangle$ where N (the set of nodes) is a set and E (the set of edges) is a subset of $N \times N$ (the set of pairs of nodes).

A subgraph of a directed graph $\langle N, E\rangle$ is a directed graph $\langle M, D\rangle$ where $M \subseteq N$ and $D = E \cap (M \times M)$. The subgraph $\langle M, D\rangle$ is defined to be a *prefix* of $\langle N, E\rangle$ iff for every edge $\langle m, n\rangle$ in E, if n is in M then m is in M.

2.5.2 C-Struct Sets

A c-struct set *CStruct* is defined in terms of an element $\perp$ ("null" element), a set of commands *Cmd*, an operator $\bullet$ that appends a command to a c-struct, and a set of axioms listed later.

A c-struct is a very general data structure that allows for different agreement problems to be represented.

Before we present the four axioms of a c-struct set, some definitions are necessary. We write a finite sequence of commands, or **c-seq** hereafter, as $\langle C_1, C_2, \ldots, C_m\rangle$. The set of all finite sequences whose commands belong to $S \subseteq Cmd$ is denoted $Seq(S)$. Notice here

that several repetitions of some element C may appear in a sequence $\sigma \in \mathit{Seq}(S)$. We define the operator $\bullet$ over the sequences of commands as follows:

$$v \bullet \langle C_1, \ldots, C_m \rangle \triangleq \begin{cases} v & \text{if } m = 0, \\ (v \bullet C_1) \bullet \langle C_2, \ldots, C_m \rangle & \text{otherwise} \end{cases}$$

We say that a c-struct w **extends** a c-struct v, or equivalently that v **prefixes** w, iff there exists a c-seq σ such that $w = v \bullet \sigma$. This fact is denoted as $v \sqsubseteq w$. Given a set T of c-structs, we say that v is a **lower bound** of T iff $v \sqsubseteq w$ for all w in T. A **greatest lower bound (glb)** of T is a lower bound v of T such that $w \sqsubseteq v$ for every lower bound w of T; we represent it with $\sqcap T$. Similarly, we say that v is an **upper bound** of T iff $w \sqsubseteq v$ for all w in T. A **least upper bound (lub)** of T is an upper bound v of T such that $v \sqsubseteq w$ for every upper bound w of T. We note it $\sqcup T$. If $\sqsubseteq$ is a reflexive partial order on the set of c-structs and a glb or lub of T exists, then it is unique. Two c-structs v and w are defined to be **compatible** iff they have a common upper bound, and a set S of c-structs is compatible iff its elements are pairwise compatible.

We say that c-struct v is **constructible from** a set P of commands if $v = \bot \bullet \sigma$, for some c-seq σ containing all the elements of P. A c-struct v **contains** some command C when v is constructible from some set P of commands such that $C \in P$. We define $\mathit{Str}(P)$ as the set of all c-structs constructible from subsets of P for some set P of commands—that is, $\mathit{Str}(P) \triangleq \{\bot \bullet \sigma : \sigma \in \mathit{Seq}(P)\}$.

A c-struct set *CStruct* must satisfy axioms CS1-CS4 below. CS1-CS2 are basic requirements to satisfy the properties discussed above. As we shall see shortly, CS3 and CS4 are necessary for Generalized Paxos and similar algorithms to ensure the safety and liveness properties of Generalized Consensus.

CS1. $\mathit{CStruct} = \mathit{Str}(\mathit{Cmd})$

CS2. $\sqsubseteq$ is a reflexive partial order on *CStruct*.

CS3. For any $P \subseteq \mathit{Cmd}$ and any c-structs u,v, and w in $\mathit{Str}(P)$:

- $\sqcap\{v, w\}$ exists and is in $\mathit{Str}(P)$.
- If v and w are compatible, then $\sqcup\{v, w\}$ exists and is in $\mathit{Str}(P)$.
- If $\{u, v, w\}$ is compatible, then u and $\sqcup\{v, w\}$ are compatible.

CS4. For any compatible c-structs $v, w \in \mathit{CStruct}$ and $C \in \mathit{Cmd}$, if v and w both contain C is in $\sqcap\{v, w\}$.

We can now generalize the original definition of consensus to deal with c-structs instead of single absolute values. This new problem is defined in terms of a c-struct set *CStruct* that includes a $\bot$ value, a set of commands *Cmd*, and an operator $\bullet$. Proposers propose commands in *Cmd* and given some learner l, we let $\mathit{learned}[l]$ be the current learned c-struct (initially $\bot$). Generalized Consensus is defined by the following properties:

Nontriviality: For any learner l, *learned*$[l]$ is always a c-struct constructible from some of the proposed commands.

Stability: For any learner l, if the value of *learned*$[l]$ at any time is v, then $v \sqsubseteq$ *learned*$[l]$ at all later times.

Consistency: The set $\{$*learned*$[l] : l$ is a learner$\}$ is always compatible.

Liveness: For any proposer p and learner l, if p, l, and a quorum Q of acceptors are non-faulty and p proposes a command C, then *learned*$[l]$ eventually contains C.

2.6 Command-Structure Set Examples

We now give some examples of some common Command-Structure Sets found in the literature (LAMPORT, 2004).

2.6.1 Command Sequences (CSeq)

The most simple example of a c-struct set is the set *Seq(Cmd)* of all command sequences, where $\bullet$ is the usual list append operator and $\sqsubseteq$ the ordinary list prefix relation. Two CSeqs are compatible iff one is a prefix of the other. From this, one can see that the set CSeq satisfies CS1-CS4.

The reader might find useful the following **lub**($\sqcup$) and **glb**($\sqcap$) examples, since there is a lack of concrete examples in the literature. Consider a set of CSeq c-structs, commands as simple integers and that no command commute with each other:

a) $\sqcup\{\langle 1,2,3\rangle, \langle 1,2,4\rangle\} = \not\exists$

b) $\sqcup\{\langle 1\rangle, \langle 1,2\rangle, \langle 1,2,4\rangle\} = \langle 1,2,4\rangle$

c) $\sqcap\{\langle 1,2,3\rangle, \langle 1,2,4\rangle\} = \langle 1,2\rangle$

d) $\sqcap\{\langle 1,3\rangle, \langle 1,2,3\rangle\} = \langle 1\rangle$

e) $\sqcap\{\langle 1,3\rangle, \langle 2,3\rangle\} = \bot$

2.6.2 Nonduplicate Command Sequences

The previous c-struct set allows multiple copies of a proposed command to appear in the command sequence. Suppose a banking system, it should not allow a single c = *withdraw $100* command to withdraw more than $100. Typically this problem is dealt with by making the commands idempotent (i.e. that can be applied multiple times without changing the result beyond the initial application). The state machine used to implement this banking system can be defined so that only the first execution of a single

withdraw command be executed. Considering that commands contain a unique identifier, c can withdraw \$200 by sending two distinct commands $c = \textit{withdraw \$100}$.

A different way of solving this problem of duplicate commands is to solve a different consensus problem that eliminates them, that is to take *CStruct* to be the set of command sequences with no duplicates, by defining the $\bot$ operator over the sequences as follows (consider that $\circ$ is the operator that appends a command to a command sequence):

$$v \bullet C \triangleq \begin{cases} v & \text{if } C \in v, \\ v \circ C & \text{otherwise} \end{cases}$$

The prefix relation and compatibility rule are kept from the CSeq c-struct set. Note that redefining the consensus problem like this does not make the problem of removing duplicate commands easier, it just transfers the problem from the state machine to the computation in the $\bullet$ operator. Instead of detecting duplicate commands while executing the state machine, one detects them while executing the consensus algorithm. This c-struct set is equivalent to the *atomic broadcast* problem (HADZILACOS; TOUEG, 1994).

2.6.3 Command Sets (CSet)

A simple agreement problem is obtained when we let *CStruct* to be the set of all finite sets of commands. The $\bot$ is represented as the empty set, $\sqsubseteq$ is the equivalent to the operation $\subset$ and $\bullet$ is defined to $v \cup \{C\}$, as in:

$$v \bullet C \triangleq v \cup \{C\}$$

This c-struct set is interesting because the resulting agreement problem is very easy to solve (this c-struct set is equivalent to the *reliable broadcast* problem (HADZILACOS; TOUEG, 1994). Proposers simply send commands to learners, and a learner l adds a command to $\textit{learned}[l]$ whenever it receives a proposer's message containing that command (LAMPORT, 2004).

We now provide some examples of **lub**'s and **glb**'s considering the same requirements in 2.6.1 but with CSet as c-structs.

a) $\sqcup\{\langle 1,2,3\rangle, \langle 1,2,4\rangle\} = \langle\{1,2,3,4\}\rangle$

b) $\sqcup\{\langle 1\rangle, \langle 2\rangle, \langle 4\rangle\} = \langle\{1,2,4\}\rangle$

c) $\sqcap\{\langle 1,2,3\rangle, \langle 1,2,4\rangle\} = \langle\{1,2\}\rangle$

d) $\sqcap\{\langle 1,3\rangle, \langle 1,2,3\rangle\} = \langle\{1,3\}\rangle$

e) $\sqcap\{\langle 1,3\rangle, \langle 2,3\rangle\} = \langle\{3\}\rangle$

2.6.4 The $\perp$ Command

The generalized consensus can be generalized to the ordinary consensus problem of choosing a single command. One can define $CStruct$ to be the set $Cmd \cup \{\perp\}$ consisting of all commands together with the additional element $\perp$. The operator $\bullet$ is defined as:

$$v \bullet C \triangleq \begin{cases} C & \text{if } v = \perp, \\ v & \text{otherwise} \end{cases}$$

From this, it follows that $v \sqsubseteq w$ iff $v = \perp$ or $v = w$. Two commands are compatible iff they are equal, and every command contais every command, since $C = C \bullet \langle D \rangle$ for any commands C and D.

With this c-struct, one can reduce generalized consensus to the ordinary consensus problem, where $learned[l] = \perp$ means that l has not yet learned a value. As an example, the consistency condition of generalized consensus asserts that for any two learners, either the values they learned are equal or one of the values is equal to $\perp$.

2.6.5 Command Histories (Chistory)

Until this point, it has been presented c-struct sets that can be used to solve problems that lack a total order of commands (e.g. CSet) or that requires a total order of commands (e.g. CSeq). But it is important to remember that to define an execution of a set of commands one does not need to totally order them; it suffices to determine the order of the pairs of commands that are non-commuting. Determining whether two commands commute can be a difficult task, so one introduces an interference relation $\asymp$ (also referenced as *dependence* relation) and requires $A \asymp B$ to hold for any pair of non-commuting commands A and B. Suppose a system that coordinates read/write operations in shared registers and command A (a read to register x) and command B (a write to register x); $A \asymp B$ could be considered true since the ordering of the commands is relevant and they do not commute.

We assume that the relation $\asymp$ on commands is symmetric, that is: $A \asymp B$ iff $B \asymp A$ for any commands A and B. The equivalence relation $\sim$ on command sequences is defined by letting two sequences be equivalent iff one can be transformed into the other by permuting elements in such away that the order of all pairs of interfering commands (commands that do not commute) is preserved. The precise definition extracted from (LAMPORT, 2004) is: $\langle A_1, \ldots, A_m \rangle \sim \langle B_1, \ldots, B_n \rangle$ iff $m = n$ and there exists a permutation π of $\{1, \ldots, m\}$ such that, for each $i, j = 1, \ldots, m$:

- $B_i = A_{\pi(i)}$
- If $i < j$ and $A_i \asymp A_j$, then $\pi(i) < \pi(j)$.

A *command history* is defined to be an equivalence class of command sequences under this equivalence relation. Command histories are isomorphic to Mazurkiewicz traces (MAZURKIEWICZ, 1985), which were introduced to study the semantics of concurrent systems.

Consider that *CStruct* now is the set of all command histories (i.e. the quotient space $Seq(Cmd)/\sim$), and we define $\bullet$ as:

$$[\langle C_1, \ldots, C_m \rangle] \bullet C \triangleq [\langle C_1, \ldots, C_m, C \rangle]$$

for any command sequences $\langle C_1, \ldots, C_m \rangle$ and command C. For any two command sequences σ and τ, if $[\sigma] = [\tau]$ then $[\sigma \circ \langle C \rangle] = [\tau \circ \langle C \rangle]$, for any command C. This uniquely defines operator $\bullet$.

In order to show that the set of command histories is a c-struct, one must note that the command history $[\langle C_1, \ldots, C_m \rangle]$ is isomorphic to a directed graph $\mathrm{G}([\langle C_1, \ldots, C_m \rangle])$ whose nodes are the commands C_i, where there is an edge from C_i to C_j iff $i < j$ and $C_i \asymp C_j$.

One can easily check that for any two command sequences σ and τ :

- $[\sigma] = [\tau]$ iff $\mathrm{G}(\sigma) = \mathrm{G}(\tau)$.
- $[\sigma] \sqsubseteq [\tau]$ iff $\mathrm{G}(\sigma)$ is a prefix of $\mathrm{G}(\tau)$.
- $[\sigma]$ and $[\tau]$ are compatible iff the subgraphs of $\mathrm{G}(\sigma)$ and $\mathrm{G}(\tau)$ consisting of the nodes they have in common are identical, and C$\not\asymp$D for every node $\langle C, i \rangle$ in $\mathrm{G}(\sigma)$ that is not in $\mathrm{G}(\tau)$ and every node $\langle D, j \rangle$ in $\mathrm{G}(\tau)$ that is not in $\mathrm{G}(\sigma)$.

The $\bullet$ operator is easier to grasp if we instantiate *CStruct* as a Mazurkiewicz trace: Recall that $\asymp$ is the binary, symmetric and irreflexive relation over the set of commands modeling that two commands are non-commuting. A command history u is a digraph $(\varepsilon_u, <_u)$, where ε_u is a subset of the set of commands, and $<_u$ is a partial order over ε_u. Let $\perp$ be an empty graph, we define the operator $\bullet$ by:

$$u \bullet C \triangleq \begin{cases} u & \text{if } C \in u, \\ (\varepsilon_u \cup \{C\},\ <_u \cup \{(D,C) : D \in \varepsilon_u \wedge D \asymp C\}) & \text{otherwise} \end{cases}$$

In similar fashion to previous c-struct sets, we now provide a concrete example of **lub** and **glbs** of Chistory c-structs represented as digraphs in a set T:

$$T = \left\{ \begin{array}{l} a \rightarrow c \quad b \rightarrow d \\ a \rightarrow b \rightarrow d \\ a \rightarrow c \rightarrow b \rightarrow e \end{array} \right\}$$

The lower bounds of T are $\{\perp, a, a \rightarrow b\}$, being $\sqcap T = a \rightarrow b$ (the greatest lower bound of T) and $\sqcup T = a \rightarrow c \rightarrow b \rightarrow d \rightarrow e$ (the least of upper bound of T).

2.7 Generalized Paxos

Paxos inherently defines a total order for client requests in a single leader environment. Establishing a total order relying on one leader is a widely accepted solution since it guarantees the delivery of a decision with the optimal number of communication steps (though the existence of an overloaded or slow leader could become a bottleneck and limit its effectiveness (CAMARGOS; SCHMIDT; PEDONE, 2006)).

Fast Paxos extends Classic Paxos by allowing fast rounds, in which a decision can be learned in two communication steps without relying on a leader but requires bigger quorums. The optimization brought by the fast execution mode of Fast Paxos can be nullified once a collision happens, in the root of collisions is also the same idea present in Paxos, the fact that defining a total order can be an unnecessary overestimation of conflicts among client requests, because the outcome of two non-conflicting requests is independent of their commit order. As a consequence, total order limits the parallelism with request execution, as all requests have to execute serially. The problem of ordering sequences of commands according to their actual conflicts has been originally proposed by Pedone et. al in the Generic Broadcast problem (PEDONE; SCHIPER, 2002), which defines a total order on only those transactions that depend upon each other.

The idea of ordering only the sequence of non-commuting commands was then improved through the formalization of the generalized consensus problem by Lamport in (LAMPORT, 2004). The result of Lamport's efforts took form in a Paxos variant named Generalized Paxos.

Generalized Paxos works in a similar way to Paxos and Fast Paxos but now the values which the agents of the protocol work with are c-structs, so the changes are made accordingly. Since the protocol is a solution to the Generic Consensus problem it also allows for a single Generalized Paxos implementation to solve different agreement problems depending on the *CStruct* set the algorithm was instantiated with.

This section details GPaxos, Lamport's solution to the Generalized Consensus problem (LAMPORT, 2004). We provide a pseudo-code description of this algorithm in Algorithm 2.1 based on the explanation provided in (SUTRA; SHAPIRO, 2010), where we describe GPaxos as a set of atomic actions. For each action, its effects (**eff**) are guarded by one or more preconditions (**pre**). A comment in square brackets indicates the role of the process, either a proposer (*Proposers*), an acceptor (*Acceptors*), a coordinator (*Coordinators*), or a learner (*Learners*).

2.7.1 Ballots and quorums

GPaxos executes an unbounded sequence of asynchronous rounds, or *ballots*. We associate a ballot with a ballot number, or *balnum*, picked in *BalNum* that uniquely

identifies it. Balnums are unbounded and they form a well-ordered set for some relation $<$, where 0 is the smallest element. In what follows, we identify a ballot with its balnum.

During a ballot, a learner attempts to learn one or more c-structs containing proposed commands. To this end, GPaxos relies on quorums of acceptors, i.e. non-empty subsets of *Acceptors*. Quorums are constructed as follows: We map to each ballot m to a set of quorums $quorum(m)$. An element in $quorum(m)$ is a quorum of m or an m-quorum. A ballot m is either *fast* or *classic* and is associated with a unique coordinator $coord(m)$ in *Coordinators*. We consider hereafter that:

Q1. For any two quorums Q and Q', $Q \cap Q' \neq \{\}$ holds.

Q2. Given a fast ballot m, two m-quorums Q_1 and Q_2, and some n-quorum Q, it holds that: $Q_1 \cap Q_2 \cap Q \neq \{\}$.

Processes participating in a GPaxos instance can compute locally the mapping of ballots to quorums and to coordinators. Such a computation may take the following form: Every process is at the same time an acceptor and a coordinator. A balnum is an integer, and ballot m is coordinated by the m^{th} coordinator (modulo $|Coordinators|$). A ballot m is fast iff m is even. If m is classic, the quorums of m are all the majority sets. Otherwise m is fast and a quorum Q should satisfy $|Q| > \frac{3}{4}|Acceptors|$ (LAMPORT, 2004).

2.7.2 Variables and further definitions

To propose a command C, a process packs C in a propose message and sends it to all acceptors and coordinators in the system (line 3).

Acceptors ensure the long-term memory of the system. They successively join ballots and vote during them. Each acceptor a maintains three variables: the current ballot (bal_a), the latest ballot during which it *accepted* (voted for) a c-struct ($cbal_a$), and the c-struct it accepted at that ballot ($cval_a$). Initially for every acceptor a, $bal_a = cbal_a = 0$, and $cval_a = \bot$.

At the beginning of ballot m, $coord(m)$ tries to convince acceptors to join m. If enough acceptors participate in m, $coord(m)$ suggests one or more c-structs. A coordinator c stores the latest ballot it started ($maxStart_c$), and the latest c-struct it suggested at that ballot ($maxTried_c$). If no c-struct has been suggested in $maxStart_c$, $maxTried_c$ equals $none \notin CStruct$. At the start of the algorithm, $maxTried_c = 0$ and $maxStart_c$ equals $\bot$ if $c = coord(0)$, and $none$ otherwise.

Considering some ballot m and a c-struct u, we say that:

- u is **chosen** at m, when there exists an m-quorum of acceptors Q, such that for every acceptor $a \in Q$, u prefixes the c-struct accepted by a at ballot m;

- ❑ u is **choosable** at m when u is chosen at m, or it might later be chosen at m;
- ❑ u is **safe** at m when it suffixes all the c-struct choosable at m.

Based on these definitions, GPaxos ensures three key invariants:

S0 If a c-structs u is learned, then u is chosen at some ballot.

S1 If two c-structs u_1 and u_2 are accepted at some classic ballot m, then $\{u_1, u_2\}$ is compatible.

S2. If an acceptor a accepts a c-struct u, then u is safe at some ballot m.

Invariants S0 and S2 together with assumption Q1 on quorums imply that learned c-structs are compatible. As a consequence, Generalized Paxos satisfies the consistency requirement of Generalized Consensus.

2.7.3 Algorithmic Details

We now detail how GPaxos executes a classic ballot to maintain invariants S0-S2.

- ❑ *phase1A*(m): When it start a ballot m, the coordinator of m, denoted hereafter c, sends a 1A message labelled m to the acceptors (line 10).
- ❑ *phase1B*(m): When an acceptor a receives a 1A message labelled m, and bal_a is strictly smaller than m, a *joins* ballot m by setting bal_a to m. Then acceptor a sends a 1B message labelled with m containing $cbal_a$ and $cval_a$ to the coordinator c (line 16).
- ❑ *phase2Start*(m, R, k): Coordinator c executes this action once there exists a ballot k and a quorum Q such that
  - *coord*(m) received a 1B message labelled m from every acceptor in Q,
  - k is the highest ballot *coord*(m) has heard of in the 1B messages so far, and
  - for every ballot n such that $k \leq n < m$, R is a quorum of n.

  c extracts a c-struct that is safe at ballot m and stores it in $maxTried_c$. Then, it suggests this to the acceptors in a 2A message (line 30).

  To make this action clearer, we extracted and adapted from (CAMARGOS, 2008) the following example:

  Suppose that c has received messages: $\langle$"1B", $i, j, v_k\rangle$ (i and j are ballots with $j < i$ and for simplicity, we assume that $i = bal_i$, $j = cbal_j$ of a_k and $v_k = cval_j$ of a_k) messages from a quorum of three acceptors $a_k, 1 \leq k \leq 3$. Also, assume for simplicity that the c-struct used is a total order of commands, easily represented by a

sequence, and that the messages had the following c-structs: $v_1 = \langle a, b, c, e, f \rangle$, $v_2 = \langle a, b, c, d \rangle$ and $v_3 = \langle a, b, c, e, d \rangle$. Observe that, since the three acceptors form a quorum, both the sequences $\langle a \rangle$, $\langle a, b \rangle$ and $\langle a, b, c \rangle$ were chosen and might have been learned. Hence, c is obliged to pick the three c-structs to use in the second phase of the algorithm. Hence, c picks the longest c-struct, whose the others are prefixes of; in fact, $\langle a, b, c \rangle$ is the longest prefix of all received c-structs.

Suppose now that there is another quorum in the system, formed by acceptors a_1, a_3, and a_4. The two quorums are represented graphically in Figure 5, below. If a_4 has accepted any sequence in round k that extends $\langle a, b, c, e \rangle$, then such a sequence was also chosen and should be picked by c. Since c never heard from a_4, it has no option but to behave safely and pick sequence $\langle a, b, c, e \rangle$. Since a_1 and a_3 disagree about which command should be the fifth in the sequence, $\langle a, b, c, e \rangle$ is the shortest sequence that extends all possibly chosen sequences.

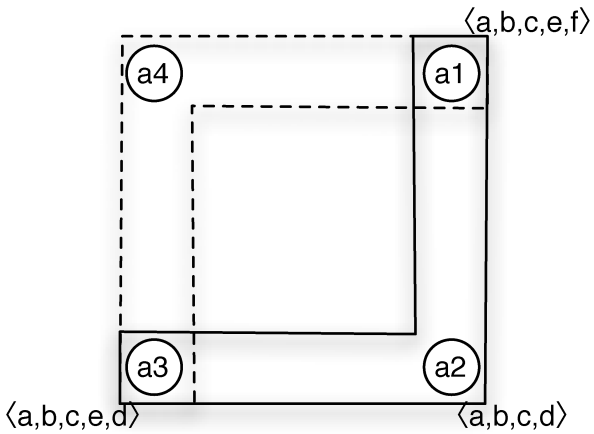

Figure 5 – A system of four acceptors and their accepted values. The two marked sets of acceptors constitute quorums.

Generalizing from the previous two paragraphs, once the coordinator receives the messages 1B from a quorum of acceptors, it picks the c-struct that extends the largest prefix of the c-structs seen (i.e, its glb), which was already chosen, and their smallest possibly chosen extensions (lub). This c-struct is then sent to the acceptors to ensure that if a c-struct v has been chosen at some round j and w is accepted by some acceptor at a higher-numbered round, then $v \sqsubseteq w$. As before, for coordinator c to gather the set of all possibly chosen c-structs in previous rounds, it suffices to look at the messages 1B with the highest-numbered *cbal* value. More formally, let k be such a round number. There are only two cases to consider.

First, if there is no k-quorum R such that, for every acceptor a in $R \cap Q$, c has received a message 1B from a with $cbal_k$, then c is assured that no c-struct has been or might be chosen at k. Moreover, the algorithm ensures that if a c-struct v has

been chosen at a round $j < k$, then any value w accepted at k satisfies $v \sqsubseteq w$. Therefore, c can pick any c-struct received in one of the messages 1B in which $cbal_k$.

If the first case does not apply, then, for every k-quorum R such that c has received a message 1B with $cbal_k$ from every acceptor in $R \cap Q$, c calculates the glb of the c-structs $cval$ received in such messages and adds it to a set γ initially empty. After that, γ will contain all c-structs that have been or might be chosen at lower-numbered rounds. The conditions previously seen ensures that γ is compatible and, therefore, has a least upper bound $\sqcup\gamma$ that can be safely picked by c.

After picking a c-struct val based on the previous two cases, c sends a message $\langle$"2A", $i, val\rangle$ to all acceptors.

- *phase2AClassic*(m, C): When $maxTried_c$ differs from *none*, by construction $maxTried_c$ is safe at m. If m is classic, c appends newly proposed commands to $maxTried_c$ and suggests the resulting c-struct to the acceptors (line 38).

- *phase2BClassic*(m, u): When an acceptor a belonging to an m-quorum receives a 2A message containing a c-struct u and a can join ballot m (or has joined it previously), a accepts u by assigning u to $cval_a$ (line 45). Acceptor a then updates $cbal_a$ and bal_a to the value of m (lines line 46 and line 47), and sends a 2B message containing $cval_a$ to the learners (line 48). Since c-struct u extends $maxTried_c$, every c-struct accepted at ballot m prefixes $maxTried_c$ (invariant S1). Moreover $maxTried_c$ is safe at ballot m, thus every accepted c-struct is safe at ballot m (invariant S2).

- *learn*(m, Q, u): A learner l learns a c-struct u once l knows that u is chosen at m (lines line 58 and line 59). To learn u, learner l assigns to $learned_l$ the value of $\sqcup \{learned_l, u\}$. This maintains the stability invariant of generalized consensus.

At first glance, GPaxos has a latency of five steps in a classic ballot, as this is the length of the causal path between propose, 1A, 1B, 2A and 2B messages. However, as long as $coord(m)$ does not crash and no coordinator starts a ballot higher than m, $coord(m)$ may suggest new commands within m. As a consequence, since $coord(0)$ can skip phase one of ballot 0 ($\bot$ is *de facto* safe at ballot 0), every command is learned in three communication steps during a nice run.

2.7.4 Fast ballots, collisions and recovery

To further reduce latency, acceptors execute *phase2BFast* during fast ballots:

- *phase2BFast*(C): Once an acceptor a has joined a fast ballot (line 51), and accepted the safe c-struct suggested by the coordinator (line 52), a tries to extend it with newly proposed commands. More precisely, when a receives a propose message

containing a command C, it sets $cval_a$ to $cval_a \bullet C$ (line 54), then sends a 2B message containing the new value of $cval_a$ to the learners (line 55).

Commands accepted during a fast ballot are learned in two steps: the causal path contains a propose message and a 2B message. A fast ballot leverages both the spontaneous ordering of the messages by the network and the compatibility of c-structs, as we illustrate below:

- ❑ Example: Let a_1 and a_2 be two acceptors that joined a fast ballot m. Suppose that a_1 and a_2 form an m-quorum, and accept u, the c-struct suggested by $coord(m)$ at ballot m. If a_1 and a_2 receive two commands C and D in this order, i.e., the network spontaneously orders C before D, then both a_1 and a_2 extend u to $v = (u \bullet C) \bullet D$. As a consequence, v is chosen at ballot m. If, however, C and D are received in different orders, e.g. a_1 extends u to $v = (u \bullet C) \bullet D$ and a_2 extends u to $w = (u \bullet D) \bullet C$, then if C and D commute $v = w$, and v is still chosen at m.

However, if the set of c-structs accepted by the acceptors is not compatible, a collision occurs. A process i detects that a collision occurs at a ballot m when the following predicate holds at i:

$$collide(m) \triangleq \exists Q \in quorum(m) : \begin{cases} \forall a \in Q : rcv_a\ (\mathsf{2B}, m, \_) \\ \neg\,(\{u \mid \exists a \in Q : rcv_a\ (\mathsf{2B}, m, u)\}\ compatible) \end{cases}$$

When a collision occurs at ballot m, GPaxos starts a higher ballot. We call this a *recovery*. The latency of GPaxos equals six communication steps when a recovery occurs: two messages during the fast ballot that collides (propose, 2B), plus four messages to recover (1A, 1B, 2A, 2B). More efficient alternatives are discussed in (CAMARGOS; SCHMIDT; PEDONE, 2006).

As we mention some phases of the TLA+ specification of GPaxos later on, we provide it completely in the Annex. It is also useful if one wants to compare it with the pseudo-code version we presented.

2.7.5 Generalized Paxos path of execution

In Generalized Paxos, a ballot i starts when the ballot coordinator executes action *phase1A*(i). Acceptors then should execute action *phase1B*(i), followed by the execution of *phase2Start*(i, R, k) by the coordinator and *phase2AClassic*(i, C) by the acceptors. After this point, the execution depends on whether i is fast or classic. If i is classic and proposers keep proposing new commands to the coordinator, then the coordinator continuously executes *phase2AClassic*(i, C) with longer c-structs, followed by acceptors executing *phase2BClassic*(i, u). If i is fast and proposers keep proposing, then acceptors execute *phase2BFast*(C) to append the proposals to their accepted c-structs. In any

case, learners continuously execute $learn(i, Q, u)$ to learn the c-struct accepted and its extensions.

2.8 Related Work

Here we present some work that relates to ours in the sense of trying to optimize Generalized Paxos, that drew inspirations from it to create new solutions to the problem of distributed consensus or that provide similar effort of diminishing the gap between theory and practice in Paxos-like algorithms.

2.8.1 Fast Genuine Generalized Consensus (FGGC)

FGGC (SUTRA; SHAPIRO, 2011) is a variation of Generalized Paxos that recovers in the optimal time of a single message delay (i.e. a single communication step), yet uses the optimal number of replicas ($2f + 1$). The algorithm is defined as *genuine*, which is a generalized consensus protocol that during a nice run gets a command learned in two communication steps considering that the set $CStruct$ is compatible.

To achieve the *genuine* property in their protocol a distinction between read and write quorums is made and with it the definition of a *centered ballot*, which is a ballot where the coordinator alone forms a read quorum. The authors proceed to show that in centered ballots where the coordinator remains the coordinator of the next ballot, it can transfer information between ballots and execute the reconfiguration phase locally, reducing recovery to two steps.

The authors argue that in reasonable conditions the recovery can be done in one step. As long as no fault occurs, FGGC access the same $f + 1$ replicas – this contrast with previous generalized consensus algorithms that are considered *genuine* which require $3f + 1$ replicas and access at least $2f + 1$ of them.

In order to recover in one step when an acceptor detects a collision, FGGC assumes that learners hear from a majority of acceptors before they execute the *learn* action and that both learners and acceptors receive messages 2B. If an acceptor detects a collision, it waits until it receives a message 2B from the ballot coordinator. The acceptor then looks at the prefixes of the commands that it accepted (by receiving commands directly from proposers when executing a fast round), takes the largest such prefix that is compatible with the message 2B, and picks their **lub**($\sqcup$). This process saves all the communication steps that would have been required to accept commands in the next ballot.

The FGGC is an important work since it does not deviate a lot from the original Generalized Paxos algorithm, which means that the optimizations proposed by the algorithm can be implemented with relative ease in our project. This could lead to some comparison tests between Generalized Paxos and FGGC in order to confirm the theoretical benefits of the algorithm.

2.8.2 Multicoordinated Generalized Paxos

Multicoordinated Paxos (CAMARGOS; SCHMIDT; PEDONE, 2007) is an extension of Paxos that allows multiple coordinators to be used concurrently to improve availability. The following are important characteristics of Multicoordinated Generalized Paxos:

- ❏ It allows for an infinite number of coordinators to be used in one execution, thus eliminating the need to recover these agents in case of failures.
- ❏ Provides a more efficient use of the stable storage since the coordinators do not write on disk (since they are not recovered) and the acceptors only write on disk once per ballot in the lacking of failures, even if a wrong suspicion of failure happens.
- ❏ It allows a better load balance between coordinators and acceptors.

Having more than one coordinator per ballot reduces, possibly to zero, the downtime when the coordinator fails at the cost of a higher risk of collision. During a multicoordinated ballot m, an acceptor accepts a c-struct u only if u prefixes the c-structs received from some m-quorum of coordinators. A collision may occur during a multicoordinated ballot m when c-structs suggested by coordinators collide. In such a case, a (higher) regular ballot is started.

This possibility of collisions makes the multicoordinated approach be more attractive if the application semantics can be taken into account. As a result, the authors applied the result to Generalized Paxos, creating the new algorithm called: Multicoordinated Generalized Paxos.

Like FGGC, the changes made by the authors to Generalized Paxos are not drastic and could be easily implemented in our implementation of the GPaxos protocol.

2.8.3 Egalitarian Paxos

EPaxos (MORARU; ANDERSEN; KAMINSKY, 2013) is a multi-leader solution to the Generalized Consensus problem, à la Mencius (MAO; JUNQUEIRA; MARZULLO, 2008). This algorithm tracks dependencies (ZIELINSKI, 2005) to deliver non-conflicting commands within a fast path of two message delays. However, in the presence of conflicts, the protocol takes a slow path of four message delays.

EPaxos may adjust the order of a message that has been already proposed, according to its dependencies, to reduce communication steps in scenarios of no conflicting proposals of dependent messages. However, EPaxos needs to build a dependency graph of received messages and execute complex tasks on this graph (e.g. calculate strongly connected components). Such operations can be significantly expensive in transaction processing, since the number of dependencies in the dependency graph can rapidly grow when a transaction's size and data contention increases.

2.8.4 Alvin

The work in (TURCU et al., 2014) describes Alvin, a system for managing concurrent transactions running on a set of Geographically Distributed Systems (GDS) (i.e. a Wide Area Network (WAN)) sites, which supports general-purpose transactions and guarantees strong consistency criteria. Through a novel partial order broadcast protocol, Alvin maximizes the parallelism of ordering and local transaction processing.

At the core of Alvin is a novel Partial Order Broadcast protocol (POB) that globally orders only conflicting transactions and minimizes the number of communication steps for non-conflicting transactions. It is based on the idea of defining the agreement of consensus on the basis of message semantics, an idea previously introduced in Generalized Consensus or Generic Broadcast. The POB encompasses a novel approach for ordering transactions' commits that overcomes the limitations of existing single leader-based solutions (i.e., Generalized Paxos) when deployed in geographically distributed systems. POB does not rely on a designated leader to either order transactions or support conflict resolution in case of conflicting concurrent transactions.

Alvin's POB has been designed to inherit the benefits of the most recent multi-leader state machine replication protocols specifically proposed for GDS such as Mencius (MAO; JUNQUEIRA; MARZULLO, 2008) and EPaxos (MORARU; ANDERSEN; KAMINSKY, 2013), and, at the same time, to overcome their drawbacks. In (TURCU et al., 2014) the authors argue that:

> Alvin's POB, like Mencius, has the advantages of defining the final order of messages on the sender nodes. Typically, this technique avoids expensive distributed decisions by determining an a priori assignment of delivered positions to messages. This approach suffers from potentially expensive waiting conditions that are needed to ensure that the delivery of a message in position p does not precede the delivery of a message in position $p' < p$. However, the authors POB, unlike Mencius, relies on a quorum of replies, instead of waiting for the information about delivered positions from all nodes. This makes POB's performance robust even in scenarios where nodes are far apart or when the message sending rate is unbalanced among nodes.

Alvin's benefits are still to be confirmed since few experimental results exist concerning its POB, the only available results are the ones from the authors themselves.

2.8.5 Paxos Made Live

In (CHANDRA; GRIESEMER; REDSTONE, 2007), the authors detail the internals of the Chubby distributed lock service. This service is built on top of a shared log that is itself implemented with the help of the Paxos consensus algorithm.

Of particular interest, (CHANDRA; GRIESEMER; REDSTONE, 2007) explain a snapshot mechanism to bound the log that is fully under the control of the Paxos, while the snapshot format is application-specific (here Chubby); and a lease mechanism. Reading the state of the Chubby service goes through the Paxos coordinator. The lease mechanism avoids to read a stale content in situations when the coordinator rotates.

2.8.6 Paxos for System Builders

The work in (KIRSCH; AMIR, 2008) tries to diminish the gap between theory and practice by presenting a complete specification of the Paxos replication protocol such that system builders can understand it and implement it. The authors evaluate the performance of a prototype implementation and detail the safety and liveness properties guaranteed by their specification of Paxos.

It contemplates details regarding mechanisms to detect failures or elect a leader and a complete pseudocode, in C-like notation, for their interpretation of the protocol.

2.8.7 Fast Paxos Made Easy: Theory and Implementation

In (ZHAO, 2015) one can see a similar effort to ours applied to the Fast Paxos protocol. The authors provide some design details for their state-machine replication framework as well as a more practical take on one of the phases of the algorithm that regards the recovery from a collision.

On detecting a collision, the coordinator should initiate a recovery by starting a new classic round. As the quorum of votes contains different values, the coordinator must be careful in selecting a value that has been chosen in a previous round, or that was in the process of being chosen. However, it's not straightforward for the coordinator to determine such a value, therefore the authors present a simpler way for this value selection rule in the coordinator. It is also important to mention that this rule for value selection is similar to the one presented in (VIEIRA; BUZATO, 2008), this gives more credibility to the approach since it was also used in another practical work.

The paper also provides data on the performance of Fast Paxos in relation to its predecessor, Paxos. Their results showed that Fast Paxos is most appropriate for use in a single client configuration, for the presence of two or more concurrent clients even in a Local Area Network (LAN) would incur frequent collisions, which would reduce the system throughput and increase the mean response time as experienced by clients. Due to frequent collisions, Fast Paxos actually performs worse than Classic Paxos in the presence of moderate to large number of concurrent clients. The authors' results mirror the ones found in (VIEIRA; BUZATO, 2013) as well.

Algorithm 2.1 Generalized Paxos – code at process i

1: *propose*(C) [proposer]
2: **pre:** $C \in Cmd$
3: **eff:** *send* $(\mathsf{propose}, C)$ *to* $Acceptors \cup Coordinators$
4:
5: *phase1A*(m) [coordinator]
6: **pre:** $maxStart_i < m$
7: $i = coord(m)$
8: **eff:** $maxTried_i := none$
9: $maxStart_i := m$
10: *send* $(\mathsf{1A}, m)$ *to* $Acceptors$
11:
12: *phase1B*(m) [acceptor]
13: **pre:** $bal_i < m$
14: $rcv_{coord(m)}$ $(\mathsf{1A}, m)$
15: **eff:** $bal_i := m$
16: *send* $(\mathsf{1B}, m, cbal_i, cval_i)$ *to* $coord(m)$
17:
18: *phase2Start*(m, Q, k) [coordinator]
19: **pre:** $maxTried_i = none$
20: $maxStart_i = m$
21: $Q \in quorum(m)$
22: $\forall a \in Q : rcv_a$ $(\mathsf{1B}, m, \_, \_)$
23: $k = max\{n < m \mid \exists a \in R : rcv_a$ $(\mathsf{1B}, m, n, \_)\}$
24: **eff:** $\mathcal{R} := \{R \in quorum(k) \mid \forall a \in R \cap Q : rcv_a$ $(\mathsf{1B}, m, k, \_)\}$
25: **if** $\mathcal{R} = \{\}$
26: $maxTried_i := u$, s.t. $\exists a \in Acceptors : rcv_a$ $(\mathsf{1B}, m, k, u)$
27: **else**
28: Let $\gamma(R) \triangleq \sqcap\{u \mid \exists a \in R \cap R : rcv_a$ $(\mathsf{1B}, m, k, u)\}$
29: $maxTried_i := \sqcup\{\gamma(R) \mid R \in \mathcal{R}\}$
30: *send* $(\mathsf{2A}, m, maxTried_i)$ *to* $Acceptors$
31:
32: *phase2AClassic*(m, C) [coordinator]
33: **pre:** $maxTried_i \neq none$
34: $maxStart_i = m$
35: $\exists p \in Proposers : rcv_p$ $(\mathsf{propose}, C)$
36: $\neg isFast(m)$
37: **eff:** $maxTried_i := maxTried_i \bullet C$
38: *send* $(\mathsf{2A}, m, maxTried_i)$ *to* $Acceptors$
39:
40: *phase2BClassic*(m, u) [acceptor]
41: **pre:** $rcv_{coord(m)}$ $(\mathsf{2A}, m, u)$
42: $bal_i \leq m$
43: $\exists Q \in quorum(m) : a \in Q$
44: $cbal_i \neq bal_i \vee cval_i \sqsubset u$
45: **eff:** $cval_i := u$
46: $bal_i := m$
47: $cbal_i := m$
48: *send* $(\mathsf{2B}, m, cval_i)$ *to* $Learners$
49:
50: *phase2BFast*(C) [acceptor]
51: **pre:** $isFast(cbal_i)$
52: $bal_i = cbal_i$
53: $\exists p \in Proposers : rcv_p$ $(\mathsf{propose}, C)$
54: **eff:** $cval_i := cval_i \bullet C$
55: *send* $(\mathsf{2B}, cbal_i, cval_i)$ *to* $Learners$
56:
57: *learn*(m, Q, u) [learner]
58: **pre:** $Q \in quorum(m)$
59: $\forall a \in Q : \exists v \in CStruct : rcv_a$ $(\mathsf{2B}, m, v) \wedge u \sqsubseteq v$
60: **eff:** $learned_i := \sqcup \{learned_i, u\}$
61:

Chapter 3

Generalized Paxos: A faithful implementation

Our Generalized Paxos implementation was done on top of a framework for state machine developed with implementations of Paxos and Collision-fast Atomic Broadcast (CFABCAST) (SCHMIDT; CAMARGOS; PEDONE, 2014). The framework, which is an open source project[1], aims at being an easily extendable and modular system and to serve as a starting point for implementations of other distributed consensus algorithms (SARAMAGO, 2016).

Even though the framework simplified the task of implementing GPaxos, it had its limitations, which delayed progress at the beginning. For example, some of its features presented bugs or were implemented in a way that made some changes rather difficult, so during our time implementing Generalized Paxos, a considerable amount of time was spent understanding the framework and the tools used to build it. Once these difficulties were overcome, the framework became, in fact, a good development platform. Our fork of the original framework is also an open sourced under free software licenses project[2], as is the client application we used to validate our system [3].

In this chapter, we present a brief analysis of the framework, the rationale behind some of its parts and how GPaxos was implemented using it.

3.1 Architecture and technology

As is the case for Paxos-like protocols, Generalized Paxos is specified in Specification Language for Temporal Logic of Actions (TLA+), a formal specification language that allows concisely expressing concurrent systems.

[1] <https://github.com/r0qs/cfabcast/>
[2] <https://bitbucket.org/tfr/gpaxossimple>
[3] <https://bitbucket.org/tfr/akka-gp-client>

Despite being precise, or exactly because of it, TLA+ specifications such as GPaxos's (LAMPORT, 2004) present themselves as a great challenge regarding real-world implementation. The translation from specification to a reliable, correct and high-performance implementation is notoriously hard, as observed in the other works of (CHANDRA; GRIESEMER; REDSTONE, 2007) and (KIRSCH; AMIR, 2008).

During the implementation of GPaxos a substantial amount of work was dedicated to solving problems that emerged from the translation of the formal specification and the assumptions it contained to real code. It is of the essence that this translation occurs in a manner that preserves its correctness. Moreover, issues with some design choices only became evident after we came across some problem that was caused by something that was not explicit enough in the specification, which forced us to refactor lots of code.

The way that these sort of problems are dealt with directly affects the algorithm's performance and therefore the application that will use it.

The protocol was implemented using the Scala programming language. Using Scala proved to be of great benefit, since the language embraces both a functional style of programming, with immutable data structures, high order functions, closures, and type-classes, as well as an object-oriented one, allowing for rapid development and concise code. Likewise, it is important to notice that Scala provides the same functionalities of the Java programming language, for it maintains a interoperability with it in the sense that both languages run in the Java Virtual Machine (JVM).

Besides Scala, we also used a set of tools known as *Akka* that provides functionalities to ease up the building of distributed systems, adopting the *actors model* for concurrency instead of the traditional combination of threads and locks.

3.1.1 The argument for the Actors Model

The traditional way of handling parallel processing or concurrent processing is through the utilization of threads. In a thread-guided development, the execution of the program is divided into a series of concurrent threads of execution, which share access to the processes' memory. Ensuring correctness in such a scenario is difficult, and problems that arise are notoriously hard to debug. Consider the *lost-update* problem, for example. Suppose that 2 processes increment the value of a shared variable *acc*, both of them recover the variable's value, increment it and store it again in the shared variable. As the operations are not atomic, it is possible that their execution occurs in different orders, resulting in different updates to the value of *acc*.

To solve problems like this one, a mechanism called *lock* is commonly used, it provides a mutual exclusion concurrency control policy (i.e only one process can access the shared resource at a time). The problem with locks is the increase in complexity a program can face when using it and new problems can arise like *deadlocks*.

Handling concurrent access to shared data utilizing synchronization mechanisms leaves the developer prone to make hard to debug errors (BUTCHER, 2014). Knowing this and that Paxos-like algorithms tend to be hard to implement properly, the framework and thus, our implementation used another model for concurrency: the actors model, that offers better support to develop fault-tolerant distributed systems.

The actors model, proposed by Carl Hewitt in 1973 (HEWITT; BISHOP; STEIGER, 1973), presents a different approach to concurrency, which avoids some problems due to the use of *threads* and *locks*. In this model, the processing is done through *lightweight threads* called *actors*, that have a mailbox used to store received messages and a predefined behavior that can be altered during execution.

Each actor is executed concurrently and, most importantly, the communication is done through message passing; there is no shared state between actors and the messages are processed atomically. The communication happens asynchronously, making no assumption about the delivery order, but only that the message will be received eventually.

Thus, the actors model provides a high level abstraction that its easier to work with than traditional methods, and that is closer to the way Generalized Paxos is defined (regarding the *roles* or *agents* of the protocol that can be seen as actors). Actors also have localization transparency, which means that the user of an actor does not need to know whether this actor resides in the same local process or in a different machine to be able to communicate with it, making the task of implementing scalable software manageable.

Some evidence to these claims can be verified in the industry through the existence and success of the Erlang programming language. Erlang is a functional programming language that was originally developed to try to mitigate some existing problems in applications that were running in vast networks, that were highly concurrent and that required high availability. The point is that Erlang employs the actors model to offer support to the construction of fault-tolerant distributed systems, being widely used in real-time systems by telecoms.

3.1.2 Akka

Both our implementation and the framework in which *Collison-fast Atomic Broadcast* was implemented, make use of the middleware Akka [4], an open source toolkit designed to simplify the construction of distributed, concurrent and fault-tolerant systems aimed at the JVM. Akka was written in Scala, but it also offers APIs to Java. It supports a wide range of concurrent programming paradigms, but it emphasizes the actors model for concurrency, inspired by Erlang's design.

The use of Akka provides our prototype with the potential to scale despite the presence of failures, in an easy manner, for the fault tolerance model adopted by Akka is inspired

[4] <http://akka.io/>

by Erlang's, commonly know as "*let it crash*". In this model, the actors are organized in a supervised hierarchy, in a way that each monitored component is monitored by another component, while its possible to reinitialize them in case of failures. This model has been used in the telecom industry for years and, hopefully, it will serve our prototype to make it a potential go-to implementation to test Generalized Consensus related work.

As expected from the actors model, the exchange of messages between actors through Akka occurs in an asynchronous manner and normally with immutable data, avoiding the need for synchronization techniques. Akka's modular structure allows for a flexible construction of distributed systems, having actors as its main module. The remaining modules provide extra features that were used in our work, as:

- ❏ Interfaces for the protocols TCP/UDP (high level abstractions for communication between actors);
- ❏ Cluster support (tools to manage group of actors distributed through a cluster);
- ❏ Failure Detector Mechanisms;

3.1.3 Project Design

The system is implemented as a set of entities (Akka Cluster), that from here on we'll reference as *replicas*. All replicas in the system have a unique address (*ActorRef*) for communication and each one is composed of a variety of modules.

The architecture (Figure 6) is similar to the one used in the CFABcast implementation, attesting to the versatility of the framework. The only modules that suffered large changes were the ones concerning the consensus algorithm (Generalized Paxos instead of Collision-fast Paxos) and our client.

Initially, each client establishes a connection with one of the replicas in the system. Each replica contains a module (Akka's cluster client receptionist) responsible for registering and managing a client's connection with an agent that performs the role of a proposer in the protocol. Although some implementations make the client *be* a proposer we do not think it is a good idea in general. Making proposers and clients two disjoint sets of processes allows simpler management. Clients are free to join, leave and crash without interfering with Generalized Paxos itself.

Each role of the GPaxos protocol (i.e. *proposers, acceptors and, learners*), replica module and the replica itself are implemented as *Akka actors*, lightweight processes that can be easily created in thousands if necessary. These actors are supervised by their own replica together, the module subject to manage the members of the cluster and the failure detector.

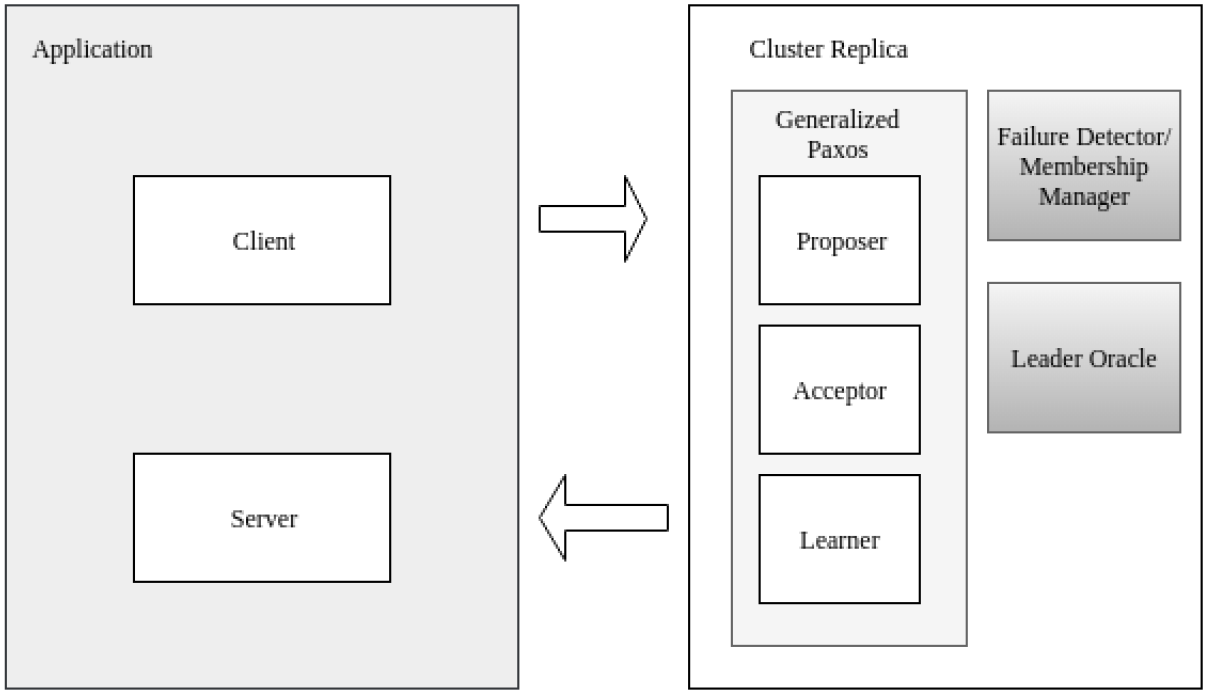

Figure 6 – Modular architecture of a Replica and its interface with the application

3.1.3.1 Explaining the Membership Manager

The membership manager is the module from the framework responsible for managing the state of the set of replicas. This problem is known in the literature as the: *Group membership* problem (CRISTIAN, 1991) and involves, for example, discovering new replicas in the system or removing replicas that may have failed. When a membership change occurs, processes can agree on which of them must complete a pending task or start a new task. The problem of reaching a consistent membership view is very similar to Consensus Problem.

The work in (CHANDRA et al., 1996) adapts the impossibility result for the Consensus Problem (FISCHER; LYNCH; PATERSON, 1985) to the Group Membership Problem. At the same time, the authors conjectured that techniques used to circumvent the impossibility of Consensus can be applied to solve the Group Membership Problem. Such techniques include using randomization (BEN-OR, 1983), probability assumptions on the behavior of the system (BRACHA; TOUEG, 1983), using failure detectors that are defined in terms of global accuracy and completeness system properties (CHANDRA; TOUEG, 1996) or gossip protocols.

In practice, the set of replicas is not subject to constant changes and a protocol that has better overall performance but that guarantees only eventual consistency might be a good choice for solving the problem. The framework and thus our project, make use of a *gossip protocol* available in Akka.

Gossiping has been shown to be effective in the detection of failures and group mem-

bership management in large scale distributed systems (SUBRAMANIYAN et al., 2006). An example of a real-world application using gossip protocols for failure detection and managing cluster members is the recent DynamoDB (DECANDIA et al., 2007) from Amazon, a high-availability key-value [5] store database.

Gossip protocols offer eventual consistency and work in a simple manner. Basically, each node periodically sends some data to a set of its neighbors chosen at random. The data is spread throughout the system from node by node (similar to how biological virus spreads in real life) until it reaches all the nodes in the system. When this happens, one might say that the protocol *converged* and therefore, all the nodes have a global view of the system that allows them to exchange information. This information can be used to inform the failure detector about the status of each node and to implement methods for discovering new nodes in the system as shown in (EUGSTER et al., 2004).

One can see all the states of a node in the cluster during a run of the system in the state diagram in Figure 7. Initially, the cluster nodes (i.e. replicas) are found in the *Joining* state and during this state, a phase of discovery of cluster members is executed through the use of the gossip protocol.

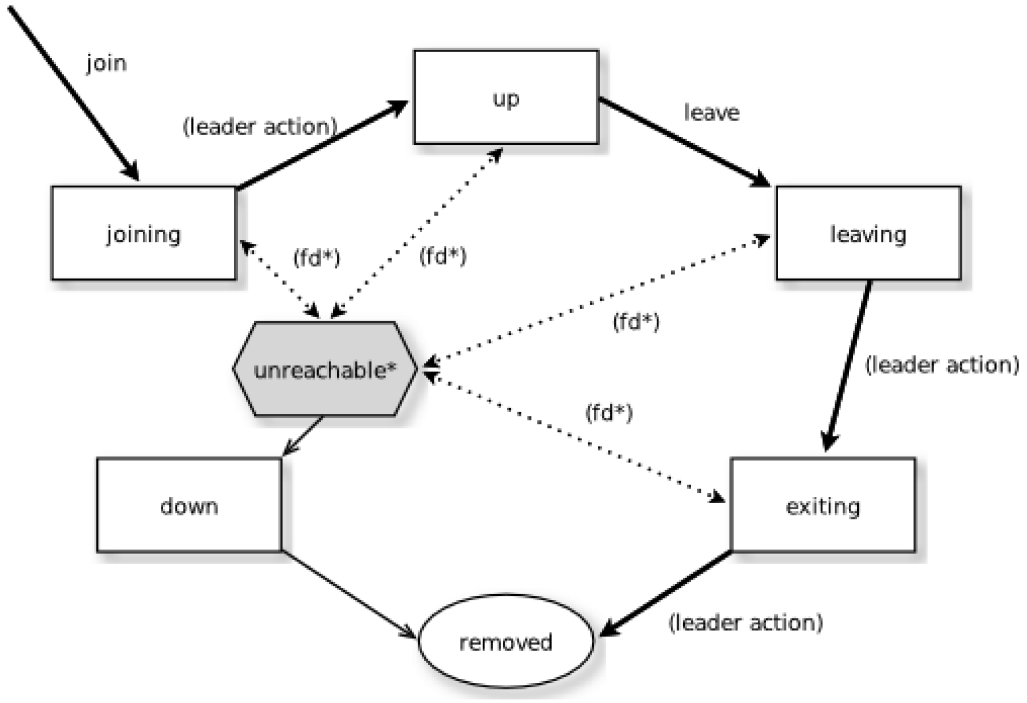

Figure 7 – State diagram of a node in the cluster (Akka team, 2016)

Once all the known nodes realize that a new node is joining the cluster (during *Joining* state), the state of that node is changed to *Up*. Considering this process of new nodes trying to join running for some time, at some point, the cluster reaches the value that

[5] A database service that provides support for storing, querying and updating collections of key/value pairs, which are identified by the key, with actual content to store as the value.

was set in the configuration files (min-nr-nodes) and the nodes initiate a leader election phase (the leader election module is notified).

There are two ways a node can be removed from the cluster: *(i)* by normal termination and changing its state to *leaving* and after *exiting* and finally being moved to the state *removed*; *(ii)* having its state altered to *unreachable* by the failure detector (transitions (fd*) in Figure 7) and then moved to *down* if its reachability is not found again.

When an actor is deemed *unreachable* by the failure detector, the failure detector module notifies the membership manager module, that after a synchrony period, removes the actor from the system and updates the configuration of the members of the cluster by disseminating this information through the network and notifying the leader election module that new leader election phase might be necessary. If a new leader is elected, then a reconfiguration phase of the protocol is initiated.

3.1.3.2 On Failure Detection

As can be seen in Figure 6, the status of each replica of the cluster is monitored by a distributed oracle (i.e. failure detector) bound to each replica.

In the framework, the membership management module is also responsible for trying to detect failures during a protocol run. For this task, it uses an implementation of the *Phi Accrual Failure Detector* (DéFAGO et al., 2004) provided by Akka.

This failure detector is interesting because instead of providing information of a boolean nature (trust vs. suspect), it outputs a suspicion level on a continuous scale. This allows it to dynamically adjusts to current network conditions the scale on which the suspicion level is expressed.

The suspicion level of a replica is named ϕ and is calculated through the Equation 1, where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat messages (disseminated through Akka's gossip protocol) inter-arrival times.

$$\phi = -log_{10}(1 - F(timeSinceLastHeartbeat)) \tag{1}$$

Based on ϕ's value, the failure detector can change a replica's state to *Up* or *Down* as seen in the Figure 7. The threshold for ϕ can be changed through a variable in the configuration files of the project. One must be aware that lower values can speed up the detection of real failures with the cost of issuing some false positives; higher values will demand higher detection time while making fewer mistakes.

3.2 On Implementing Generalized Paxos

In the next sections we provide important details on the implementation of the abstractions of Generalized Paxos Algorithm.

3.2.1 A Note on TLA+

Before providing some of GPaxos's implementation details, it is important to understand how TLA+ relates to Paxos-like algorithms. TLA+ is a formal specification language developed by Leslie Lamport (LAMPORT, 2002) the author of Paxos, so his perspective on how concurrent systems should be specified are related. From inception, Paxos and similar algorithms are described using TLA+ or something that resembles TLA+ (in the case of the original Paxos article (LAMPORT, 1998)) and not some sort of pseudocode that mirrors real life programming languages. TLA+ is based on the idea that the best way to describe things formally is with simple mathematics, and that a specification language should contain as little as possible beyond what is needed to write simple mathematics precisely. Lamport's approach when specificating Paxos was not to fully describe its algorithmic intricacies but to achieve correctness regardless of the programming language that would be used to implement it. In (LAMPORT, 2002), Lamport's ideas regarding specification of systems and the TLA+ language are left quite clear with comments like:

> Some readers may need reminding that numbers are not strings of bits, and $2^{33} * 2^{33}$ equals 2^{66}, not *overflow error*.

Or in:

> A system specification consists of a lot of ordinary mathematics glued together with a tiny bit of temporal logic.

TLA+ enables one to generate specifications written in a formal language that combines logic and mathematics, that carry a precision that may uncover design flaws before system implementation is underway. TLA+ specifications are amenable to finite model checking, that is, the model checker finds all possible system behaviors up to some number of execution steps, and examines them for violations of desired invariance properties such as safety and liveness.

Algorithms described as TLA+ specifications have no detailed instructions on how to implement them, so a big challenge while implementing Paxos-like algorithms is how one deal with mathematical aspects of the specification, for example, generating a set that contains only the elements that obey certain conditions. Depending on the condition, the complexity of a naive translation from TLA+ to code can be of no practical use without heavy optimizations. We show a practical example of this argument in Section 3.2.9.

For having such a proximity with mathematics, the use of functional programming languages (Scala in the case of our project) to translate TLA+ specifications to real projects confers the user with some guarantees regarding the correctness, since fewer abstractions from TLA+ to the code are necessary.

3.2.2 Implementing the ballot abstraction

Like ordinary Paxos, the generalized Paxos algorithm executes a sequence of numbered ballots to choose values. In GPaxos, if a ballot does not succeed in choosing values because of a failure (this includes unrecoverable collisions), then a higher-numbered ballot is executed.

In (LAMPORT, 2004), Lamport states

> We assume an unbounded set of ballot numbers that are totally ordered by a relation $<$, with a smallest ballot number that we call 0. The natural numbers is an obvious choice for the set of ballot numbers, but not the only one.

As such, we implemented ballots, or rounds as they are often called in the literature, in a straightforward manner. The ballot is abstracted through the use of a *case class*, which are just regular classes that are immutable by default, decomposable through pattern matching and succinct to instantiate and operate on. A ballot is instantiated with the following set of parameters: *(i)* the identifier of the ballot (an integer), *(ii)* a set of coordinators (the use of the data structure *Set* instead of only the reference to one coordinator, is to allow for some future optimizations regarding the availability of coordinators of the protocol) and *(iii)* a simple boolean variable used to indicate if the ballot is *classic* or *fast*.

Some argument could be made that our implementation is too simplified in the sense that it does not maintain any kind of information about which quorum of acceptors was formed in some round, but we decided to keep it simple and deal with quorum formation in a different way that will be fully explained later. We illustrate the aforementioned requirements and attest its beneficial simplicity with our code (Algorithm 3.1) extracted directly from the project. The use of a `case class` to express the class round, provides us with a certain level of safety when using Akka to send messages that have a round as a parameter.

Algorithm 3.1 – Ballot implementation

```
1  case class Round(count: Int, coordinator: Set[ActorRef], roundClassic: Boolean)
2                  extends Ordered[Round] {
3
4    //Result of comparing this with operand that.
5    //returns x where x < 0 iff this < that x == 0 iff this == that x > 0 iff this > that
6    override def compare(that: Round) = (this.count - that.count) match {
7      case 0 if (!coordinator.isEmpty) =>
8                (this.coordinator.head.hashCode - that.coordinator.head.hashCode)
9      case n =>
10               n.toInt
11   }
12
13   def inc(): Round = Round(this.count + 1, this.coordinator, true)
14
```

```
15    //True stands for Classic Round, otherwise its a Fast Round
16    def isClassic(): Boolean = this.roundClassic
17
18    override def toString = s"<␣" + count + ";␣" + {
19      if(coordinator.nonEmpty)
20        coordinator.head.hashCode else coordinator }
21      + ";␣" + {
22      if(this.isClassic)
23        "Classic"
24      else
25        "Fast"} + "␣>"
26  }
27
28  object Round {
29    def apply() = new Round(0, Set(), true)
30  }
```

In the configuration files of our implementation, there is no policy specifying how GPaxos will intercalate the use of fast and classic rounds. Being pragmatic, we added a special kind of message one can send to the coordinator asking it to initiate a new ballot with a different type, the only enforced rule is that the first ballot is a classic one.

This is an interesting point to investigate in the protocol, since (LAMPORT, 2004) does not fully explore on how to alternate the types of ballots during execution and how this can impact the performance of the protocol. We argue that some kind of oracle that leverages network monitoring could be used to decide whether to switch between fast or classic execution. For example, considering that the network is not overloaded, then spontaneously ordering of messages is likely and, as a consequence, collisions are less probable, making the use of fast ballots more favorable. However, if the network is overloaded, then there is no reason for the protocol to keep trying to fast execute, for the state of the network can increase the probability of collisions in a way that the time spent recovering from them will overshadow the optimization provided by the fast execution.

3.2.3 Command abstraction

There is little in the literature concerning exactly how commands (in GPaxos wrapped in a proposal) should be represented, so we chose the safe way and made it as generic as possible, with certain characteristics that would ease up some requirements of the protocol.

A command is represented as an *abstract class* with one variable called *content* and an abstract method *commuteWith*. The *content* variable is of a generic type, it allows the modeling of a simple commands as an integer value, or a more complex one for prototyping a banking system (e.g. “client1 **withdraws** $100”). The *commuteWith* method outputs a boolean indicating if two commands are commutable.

Knowing if two commands commute can be a hard task (LAMPORT, 2004), (LAMPORT et al., 2005), thus for the prototype, we abstract this complexity by expecting the user of the protocol to define its own *commuteWith* method for each concrete com-

mand class being implemented. For the proof-of-concept it was implemented the concrete command class *Commutative Command Int* which holds as *content* the type *Integer*. Its *commuteWith* method works by comparing the two *contents* of each command and returning the boolean *true* if both commands are even or if both commands are odd integers.

3.2.4 On implementing *CStruct* Sets

One of the most important parts of the process of implementing GPaxos is how to implement the abstract construct of a command-structure set correctly. As mentioned earlier in section 2.5, a c-struct set *CStruct* is defined in terms of an element $\perp$, a set of commands *Cmd*, an operator $\bullet$ that appends a command to a c-struct, and a set of axioms. In an effort to be as close as possible to the mathematical specification, the *abstract class Cstruct* contains only the essential variables and methods that one would have to implement in order to have a correct command-structure set.

This is shown in Figure 8 where the set of commands is translated to the mutable variable *value* (has a generic type), the append operator $\bullet$ is represented as the polymorphic function *append* and the empty c-struct $\perp$ is represented with a combination of the container *value* being empty and the method *isBottom*. Moreover, its essential that a C-struct Set have a prefix relation defined (method *isPrefixOf*) and how the operations **lub** and **glb** will apply, since they are used throughout the protocol.

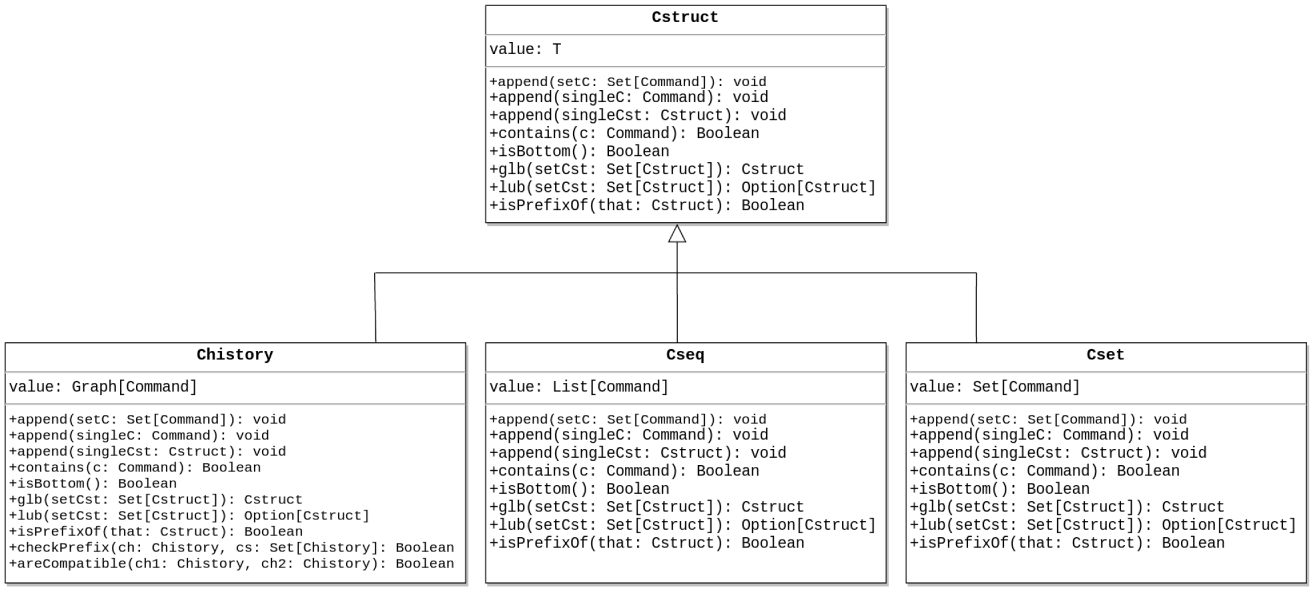

Figure 8 – UML diagram for the c-struct abstraction

The design follows a more classically object-oriented structure with mutable states. The design could see some improvement in a future work in the form of a refactor in a more functional style. For example, the *append* method could output a copy of the current c-struct with the new command appended instead of changing the content of the

object that invoked the method, this could help to avoid errors caused by mutable states and it brings the implementation closer to its formal definition.

Three basic *CStruct* sets presented in 2.5.2 were implemented in the project:

- ❏ Total order of commands (Command Sequence or CSeq) considering no repetition, its equivalent to the *atomic broadcast* problem (HADZILACOS; TOUEG, 1994);
- ❏ No order of commands (Command Set or CSet), equivalent to the *reliable broadcast* problem (HADZILACOS; TOUEG, 1994);
- ❏ Partial order of commands (Command History or CHistory), equivalent to the *generic broadcast* problem (PEDONE; SCHIPER, 2002);

3.2.5 CSeq and CSet

The implementation of CSet is about 60 LOC and CSeq 80 LOC and no big challenges were faced while creating them, as one just needs to follow the definitions closely. However, one must take into account that finding if two c-structs are equal can be a bit of an issue if one is careless with its code. Equality tests are not as trivial as one might think (ODERSKY; SPOON; VENNERS, 2011); writing a correct equality method is surprisingly difficult in object-oriented languages. In fact, after studying a large body of Java code, the paper in (VAZIRI et al., 2007) concluded that almost all implementations of equals methods are faulty.

The naive approach of just comparing the variable *value* in both c-structs, will cause errors when using c-structs as elements of some data structures from Scala (e.g. Set or LinkedHashSet), that use the underlying hash code of the objects for the equality testing. So to build our comparison test, we have overwritten the *equals* method (present in every Scala object) in a way that it leverages both the hash code and the *value* variable.

The *value* of a CSeq is of type List of Commands; this way there is a guarantee on the total order of the commands inherently from the structure (i.e. an append on a list places the element right before the last one). Considering the CSet command-structure, *value* is of type: Set of Commands, which provides the property of non-repetition of elements with no guarantees in respect to the order.

The implementation of the *append* and *contains* operations to both the c-structs are simply wrappers to the append/insertion/contains functions already present in the data structures List and Set. The interesting part lies in the implementation of the **lub** and **glb**, which differs a bit.

By the definitions and examples in Sections 2.6.1 and 2.6.3, one can notice that the **lub** and **glb** are equivalent to finding respectively the **union** and **intersection** of sets when the c-struct are CSet. In the case of CSeq, the **glb** is equivalent to the procedure of

finding the greatest common prefix of the set of c-structs and the **lub** is the operation of finding the c-struct of which all the others are prefix (i.e. the smallest common extension).

The complexity for both CSeq operations is $O(nm)$, where n is the size of the c-structs and m the size of the set of c-structs. This is so, because both algorithms work in a similar fashion, taking the **glb** for example, one iterate over the first c-struct, takes the first value and seek if this value is present in the exact same order over all the other c-structs of the set. If it finds the value in the same order in all c-structs, then it belongs to the greatest common prefix. Otherwise, the algorithm comes to a halt and outputs the previously cached prefix. The procedure is analogous in the **lub** algorithm.

3.2.6 CHistory

To our knowledge, our implementation of the Command History *CStruct* set is the only one actually using a graph data structure for the variable *value*, each Command is wrapped inside a node of a Directed Acyclic Graph (DAG). This can be useful to a client in the sense that he can choose which possible sequence of commands (acquired through searches in the graph) is more beneficial to his needs. The implementation of the CHistory is about 200 LOC and it proved to be more challenging than initially expected since there is no pseudocode description to the operations of **lub** and **glb** when c-structs are histories implemented as graphs, one has to infer them from the definitions.

It's important to notice that using a graph library [6] and some functional programming features of Scala, allowed for such a human readable code. As an example, the code for the *isPrefixOf* operation is in Algorithm 3.2 follows the formal definition for finding a graph prefix in Section 2.5.1.3.

Algorithm 3.2 – Command history prefix function

```
1   //if this is prefix of that, returns true
2   override def isPrefixOf(that: Cstruct): Boolean = {
3     val thisGraph = this.value
4     val thisNodes = thisGraph.nodes
5     val thatGraph = that.asInstanceOf[Chistory].value
6     var ans = true
7     breakable
8     {
9        thisNodes.map(n =>
10       {
11         val intersectNode = thatGraph find n;
12         if(intersectNode == None) {
13           //its not a subgraph
14           ans = false
15           break
16         } else {
17           //return the set of nodes that are direct
18           //predecessor and take the values inside the nodes
19           //to compare if both sets are equal
20           val thisIncoming = n.diPredecessors.map(e => e.value)
```

[6] <http://www.scala-graph.org/>

```
21              val thatIncoming = intersectNode.get.diPredecessors.map(e => e.value)
22              if(thisIncoming == thatIncoming) {
23                ans = true
24              } else {
25                ans = false
26                break
27              }
28            }
29          })
30      }
31      ans
32    }
```

The **glb** (Algorithm 3.3) for a CHistory as a graph works as follows: In a greedy approach we assume that the greatest prefix at the worst case will be the greatest c-struct (i.e. the graph with most nodes), we try to build the greatest prefix incrementally by taking the sequence of commands that gave origin to the greatest history. Then, we iterate over this sequence of commands and at each iteration create a c-history (i.e. subgraph) with the old state plus the current command, we proceed to check if this c-history is prefix to all the others in the set. The algorithm has time complexity of $O(n^2m)$, considering n as the size of the c-structs and m the size of the set of c-structs, due to line 8 that is $O(nm)$ and for it being repeated $O(n)$ times in line 6.

Algorithm 3.3 GLB CHistory

```
1:  Task ⊓(cs : Set[CHistory])(CHistory)
2:      greatestHistory := findGreatestHistory(cs)
3:      graphSequence := extractCommandSequence(greatestHistory)
4:      answer := ⊥                          // Answer starts as the empty CHistory
5:      param := ⊥
6:      for Cmd → graphSequence do
7:          param := param • answer • Cmd
8:          if checkPrefix(param, cs) = true then
9:              answer := answer • Cmd
10:         else
11:             return answer
```

The **lub** algorithm works in a different way, we reduce the set of CHistory c-structs through the function *areCompatible*, that is, we start with the first and second c-structs of the set and check if they are pairwise compatible, if it is the case, we make the union of the two c-structs and use the result as the input to the next comparison. As the crucial part is the *areCompatible* function, we provide the pseudocode for the formal description (2.6.5) in Algorithm 3.4.

Algorithm 3.4 Pairwise compatibility check for CHistory

```
1: Task areCompatible(ch1 : CHistory, ch2 : CHistory)(Boolean)
2:     graph1 := ch1.value
3:     graph2 := ch2.value
4:
5:     g1Cmds := graph1.getCommandsAsSet
6:     g2Cmds := graph2.getCommandsAsSet
7:
8:     commuteC := g1Cmds − g2Cmds
9:     commuteD := g2Cmds − g1Cmds
10:
11:    firstRequirement := true
12:    for C → commuteC do
13:        for D → commuteD do
14:            if C.commuteWith(D) = false then
15:                firstRequirement := false
16:                goto firstCondition.
17: firstCondition:
18:    if firstRequirement = false then
19:        return false
20:    else      // iff the subgraphs of ch1 and ch2 consisting of the nodes their have in
       common are identical
21:        nodesInCommon := g1Cmds ⋂ g2Cmds
22:        subg1 := findSubgraph(graph1, nodesInCommon)
23:        subg2 := findSubgraph(graph2, nodesInCommon)
24:        if subg1 = subg2 then
25:            return true
26:        else
27:            return false
```

Regarding the *append* method, we just iterate over the set of nodes of the graph *value* and create a directed edge from the current command to the new command iff the two commands **do not** commute, thus representing the interference relation described in 2.6.5.

3.2.7 Implementing the Acceptor

Although the acceptor is the simplest component of the protocol, it can be one of the hardest to implement due to the necessity of maintaining the acceptor's state in secondary storage. As our time to implement the protocol was quite limited, considering the amount of work that must be put to build a complete state machine replication system that utilizes Generalized Paxos, we had to make a concession regarding one need of the protocol, which is that the acceptor must have stable storage to guarantee that the implementation is *Crash-recovery* (i.e. an acceptor that crashes can rejoin the system later since its old state is not lost). As we did not implement stable storage in the acceptor, we

must consider our implementation to be *Crash-stop* and as future work, we shall expand it to meet the criterion needed to consider it *Crash-recovery*.

It is very important to think this requirement through, because the stable storage profoundly impacts the overall performance of the protocol since the acceptors state grows to infinity, in Paxos a solution to this problem is to use *Snapshots* and to Generalized Paxos the solution involves considering that the protocol will use something called: *Checkpoints*, a mechanism that allows for a c-struct to have its prefix size diminished after a certain amount of time, this idea is explored a bit further in 3.2.10.1

Regarding the acceptor's design and code, the use of Scala and Akka allowed us to produce small-sized and idiomatic code with about 130 Lines of Code (LOC), that mirrors the TLA+ specification presented in (LAMPORT, 2004) rather well. The architecture of the acceptor is similar to the other agents of the protocol. Every agent has some set of variables (as shown in the Algorithm 2.1) that we treat as a state of a state machine. In the case of the acceptor we have: *mbal[a], bal[a], val[a]* that in our code are respectively: *rnd, vrnd, vval*. The state is represented as a Scala *case class* (in this case named *AcceptorMeta*).

Algorithm 3.5 – Acceptors state as a case class

```
1  case class AcceptorMeta(
2    //following exactly the TLA+ specification
3    rnd: Round, //mbal[a]
4    vrnd: Round, // bal[a]
5    vval: Option[CStruct] // val[a]
6  )extends Serializable
```

When a new message is received by an actor, it gets atomically processed by one of that agent's functions, that is responsible for matching what type of message was received (in the case of the acceptor, messages of the types: propose, 1A, 2A) to the appropriate function responsible for processing it. This function will then process the message and as a result, change the acceptor's state (in this case *AcceptorMeta*) accordingly.

As an example, the Algorithm in 3.6 is only executed when an acceptor receives a message of type 1A. It is possible to note that this function is a clear transliteration from the specification of its TLA+ equivalent in Figure 9.

Algorithm 3.6 – Acceptor's phase1B

```
1    def phase1B(actorSender: ActorRef, msg: Msg1A,
2                state: AcceptorMeta, config: ClusterConfiguration): AcceptorMeta =
3    {
4      if (state.rnd < msg.rnd && (msg.rnd.coordinator contains actorSender)) {
5        val newState = state.copy(rnd = msg.rnd)
6        actorSender ! Msg1B(id, newState.rnd, newState.vrnd, newState.vval)
7        newState
8      } else {
9        actorSender ! UpdateRound(state.rnd)
10       state
11     }
12   }
```

$$
\begin{aligned}
&Phase1b(a,\ m) \triangleq \\
&\quad \land\ [type \mapsto \text{“}\mathsf{1a}\text{”},\ bal \mapsto m] \in msgs \\
&\quad \land\ mbal[a] < m \\
&\quad \land\ mbal' = [mbal \text{ EXCEPT } ![a] = m] \\
&\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“}\mathsf{1b}\text{”},\ bal \mapsto m,\ acc \mapsto a, \\
&\qquad\qquad\qquad\qquad vbal \mapsto bal[a],\ vote \mapsto val[a]]\} \\
&\quad \land\ \text{UNCHANGED } \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ bal,\ val\rangle
\end{aligned}
$$

Figure 9 – TLA+ specification for Phase1B

3.2.8 Implementing the Proposer

The proposer is probably the most complex agent amongst the others, and it is comprised of about 530 LOC in our implementation. Even though the proposer role shares a similar structure as the acceptor and learner, in the sense of how it receives and process messages, one must be extra careful while implementing it for reasons that will be explained.

As mentioned in 3.1.3, our clients send proposals to the proposers rather than having clients that *are* proposers; this allows for a more modular implementation.

A special proposer plays the role of the *leader* (i.e. *coordinator*) and its working is similar to the one found in classic Paxos when GPaxos is running in classic rounds (i.e. coordinator centralize proposals). As explained earlier, during fast rounds the coordinator is bypassed and the proposals are sent directly to the acceptors, in this case, the coordinator is only responsible for the reconfiguration of the systems in the presence of failures.

As well as the implementation of CFABcast, we used the detector provided by Akka as a base implementation for our leader election. A simple deterministic mechanism that chooses the smallest identifier among the proposers in the system (information acquired through the *membership manager* module of the framework) is used for leader election. This is possible due to the fact that each replica has a unique identifier. Once the leader is selected, the proposer plays this role until there is suspicion of its failure by the failure detector.

Thus, as the coordinator is also a proposer, extra care must be taken while implementing certain phases of the algorithm that only the coordinator must execute (for example, *phase2AClassic(m,C)*).

3.2.8.1 Proposer action phase2Start optimization

One of the hardest phases to understand one can find in the proposer and that can pose problems for a performant implementation, is the *phase2Start*(m, R, k). As explained in 2.7.3, in case of coordinator failure or collision at the acceptors, preventing learning of

new commands, a new ballot is started by re-executing actions *phase1A*, *phase1B*, and *phase2Start*. One of the steps performed by *phase2Start* is to determine the set $\mathcal{R}$ of k-quorum R such that, for every acceptor a in $R \bigcap Q$, it has received a 1B from a reporting acceptance of a value in k ($cbal_a = k$).

While enumerating all k-quorums and then checking if they must be in $\mathcal{R}$ works, it is not the best approach when a quorum is defined simply as a subset of the acceptors of a certain size. In this case we propose to directly enumerate the intersections between Q and possible R. Since there are multiple R that intersect with Q in the same way, the set of intersections is smaller than $\mathcal{R}$ and cheaper to calculate. If $Q' = \{a : a \in Q,$ coordinator received 1B with $cbal_a = k\}$, $QSize(b)$ is the size of a quorum for ballot b, and the new ballot is l, the resulting set of intersections is the set of all sets of acceptors with minimum size of $(QSize(k) + QSize(l)) - |Acceptors|$ and maximum size equal to the total number of messages 1B with $cbal_a = k$, that are subset of Q':

$$\{i : i \subseteq Q', |i| \geq (QSize(k) + QSize(l)) - |Acceptors|\}$$

3.2.9 Implementing the Learner

One might think that learners would be the easiest agents to implement since their only action is $learn(m, Q, u)$ (presented here in its TLA+ specification in Figure 10). In fact, it may be quite the opposite, the action serves as a good example of the gap between theory and practice in the GPaxos protocol. This can be asserted by analyzing the complexity of the patent submitted by Generalized Paxos creator in (LAMPORT et al., 2005), describing some mechanisms for commutativity detection that takes place in the learner.

$$
\begin{array}{l}
Learn(l,\ v)\ \triangleq \\
\quad \wedge\ \ \exists\, m \in BalNum : \\
\qquad \exists\, Q \in Quorum(m) : \\
\qquad\quad \forall\, a\ \ \in Q : \exists\, msg \in msgs :\ \wedge\, msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, msg.bal\ \ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, msg.acc\ \ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, v \sqsubseteq msg.val \\
\quad \wedge\ \ learned' = [learned \text{ EXCEPT } ![l] = learned[l] \sqcup v] \\
\quad \wedge\ \ \text{UNCHANGED } \langle propCmd,\ msgs,\ maxLdrTried,\ curLdrBal,\ mbal,\ bal,\ val \rangle
\end{array}
$$

Figure 10 – TLA+ specification for the Learn action

A learner agent is comprised of about 140 LOC and it works in similar fashion to the acceptor and proposer. It expects messages of type 2B and, upon receiving them, executes function *learn* that alters *LearnerMeta* (i.e. learner's state) properly.

A priori, the learner has no round as of its TLA+ specification, but to keep things simple we added a round variable (*lrnd*) to the learnersActor, as seen in (LAMPORT et al., 2005):

> A learner comprising: a memory storing a transactions table; a processor;a receiver executing in the processor and configured to receive a vote for a command from one of a plurality of acceptors, wherein the vote is associated with a ballot; a ballot updater executing in the processor and configured to determine if the ballot associated with the vote is newer than a current ballot, and to replace the current ballot with the ballot associated with the vote if it is determined that the ballot associated with the vote is newer than the current ballot;

Having a round variable at the learner makes the learner code simpler (we try to form quorums only to the greatest known round) and does not sacrifice correctness (we still fulfill the conditions of the TLA+ specification present in Figure 10).

One problem found when translating the TLA+ in Figure 10 to real code, is that the following kind of scenario can happen, supposing that:

- ❏ GPaxos is executing a fast round m;
- ❏ Instantiated with the Chistory $CStruct$ set;
- ❏ Command A commutes with C (A command is defined to commute with another command if the result of executing the two commands is independent of the order that the commands are executed.);
- ❏ Ballot m has 3 acceptors, as m is fast a quorum Q and should satisfy $|Q| > \lceil \frac{3}{4} |Acceptors| \rceil$, thus $Q = 3$;

Under these settings, while sending proposals to the acceptors, some of them accepted different but possibly compatible c-structs. Also, consider that some messages 2B got delayed while being delivered to learners, enabling some learners to receive a message 2B containing a c-struct $\langle A, C \rangle$ before receiving the message containing a previously accepted c-struct $\langle A \rangle$. A good implementation of the *learn* action must be resilient towards this kind of scenario.

Considering the scenario described, illustrated in Figure 11, we can only conclude that Learner L_1 will learn c-struct $\langle A \rangle$. This is so since c-struct $\langle A \rangle$ is the greatest prefix accepted by the quorum $\{A_1, A_2, A_3\}$. Command C will only be learned if the learner L_1 receives a new message containing a c-struct that contains C and that allows for a new quorum to be formed.

L_1 must remain correct despite the fact that the delayed messages containing c-structs $\langle A \rangle$ and $\langle C \rangle$ will arrive. It is important to notice that a learner L_n can receive messages

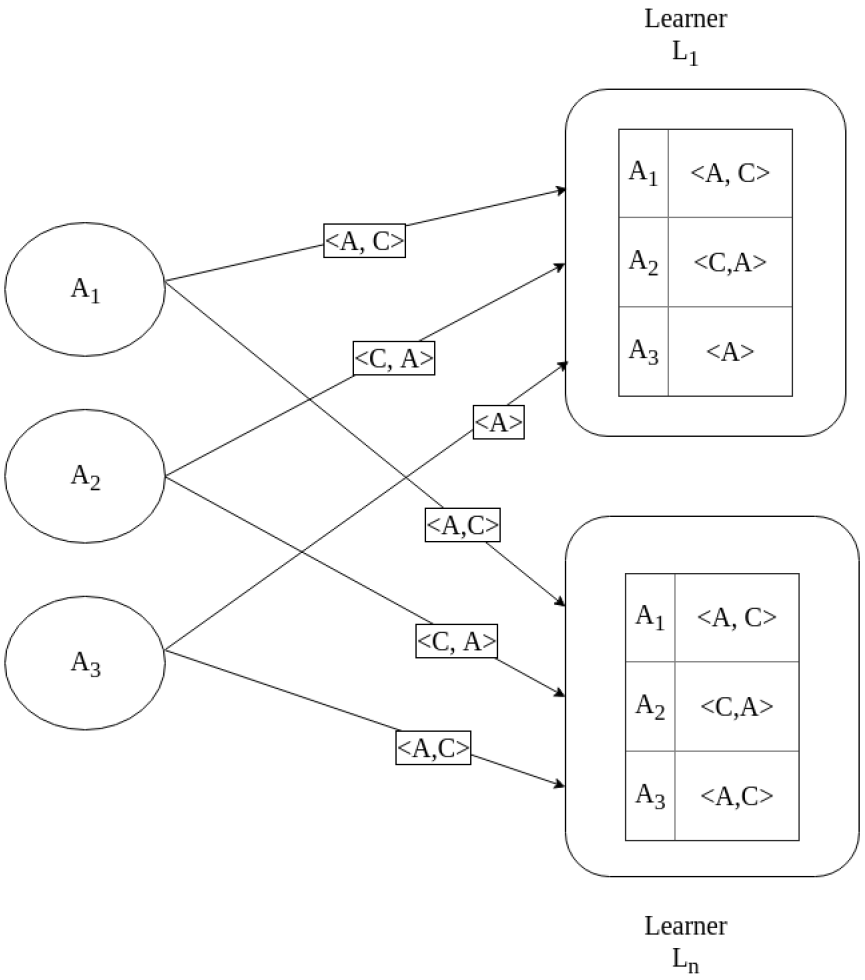

Figure 11 – Learners receiving messages 2B in different orders from 3 acceptors in a Fast Round and storing the c-structs (votes) in the *quorum2bVotes* structure.

in a different order that allows for a different quorum to be formed and thus, a potentially different value to be learned. Differently, from the normal distributed consensus problem, this is not a problem in generalized consensus, in which one must only guarantee that the values learned are compatible.

The TLA+ specification neither provides a clear way to deal with this kind of corner cases nor should it do so, since the advantage of the formal specification lies rightfully in the fact that any kind of higher-order pseudocode can be derived from the TLA+ specification as long as it maintains the correctness. We provide a more detailed solution in Algorithm 3.7 and that was implemented in our project.

3.2.9.1 The naive learning approach

The solution in Algorithm 3.7 is derived from our study and the writings in (LAMPORT et al., 2005). Lamport's patent suggests that

> In order to determine if commands commute with one another, the learner

is desirably able to pre-execute each command at a transactional substrate associated with the state machine and determine if any two commands commute based on the results of the pre-execution.

In our implementation, we modeled the commutation of commands in the abstraction of a command and we leverage this fact when calculating the append operation, the **lub** and **glb** when c-structs are Chistories.

Algorithm 3.7 Naive learning

```
1: Task learn(msg : 2B, qSize : Integer, state : LearnerMeta)(LearnerMeta)
2:                                   // Pre-conditions, assume that the learner has the following:
3:     ∧possibleQuorums : List[(List[acceptorsIds], boolean)]
4:     ∧quorum2bVotes : Dictionary[acceptorIds → cstruct]
5:     if possibleQuorums = ∅ then
6:                                                  // Generate combinations only once and set all to true
7:         possibleQuorums := CombinationsList(acceptorIds, qSize)
8:     if quorum2bVotes.size ≥ qSize then
9:         // Filter combinations using lub(⊔) so that we glb(⊓) only the compatible, set the
   non-compatible to false
10:        onlyCompatible := FilterCompatible(possibleQuorums)
11:        possibleQuorums := onlyCompatible
12:        if checkCompatible(possibleQuorums) = true then
13:                                  // Find the glb(⊓) of the quorums that have compatible C-structs
14:            toBeLearned := glbPossible(possibleQuorums)
15:            newState := state ⊔ toBeLearned
16:            return newState
17:        else
18:                                // No compatible quorums found, initiate recovery sending message to
   coordinator to initiate a new round
19:            SendMsg(CollisionRecovery, Coordinator)
20:            return state
21:    else
22:        return state                                          // Still did not reach quorum
```

Every learner keeps a structure called *quorum2bVotes* responsible for maintaining a mapping from acceptors to their votes (c-structs) in a certain round. When a learner receives a 2B, it checks if it is necessary to update the *lrnd* to the round of the received message, which will happen if, and only if, the round contained in the message is higher than the current *lrnd*, if this is the case then, the *quorum2bVotes* structure must also be updated since its information are only worth per round. After updating the round, it inserts the acceptors vote (i.e. c-struct) in the *quorum2bVotes*, one must be aware that since we cannot guarantee any kind of order about this message, we must check if we already haven't stored a more updated version of this vote (e.g. a learner has received $\langle A, C\rangle$ and only then the prefix $\langle A\rangle$, one must keep only $\langle A, C\rangle$).

When these initial conditions are met, the action *learn* in Algorithm 3.7 is executed. The Algorithm works as follows: in the first step it generates the combinations of all

possible quorums (line 7), and initially assign the boolean *true* to each one of them, to indicate that the quorum is still compatible. This approach is inspired by the idea in (LAMPORT et al., 2005):

> The learner may further improve the efficiency of conflict detection by first generating the possible acceptor quorum combinations in some fixed order, for example. The learner may then consider the combinations, in turn, starting from the beginning of the order. Once a compatible combination is detected, the learner desirably records the compatible combination for use later.

The algorithm proceeds by checking if there is a quorum of votes in *quorum2bVotes* (line 8) and, if it is the case, filters (line 10) the possible quorums by assigning the boolean *false* to the combinations that are incompatible (i.e. combinations that have no **lub** of its c-structs). If there are still compatible quorums, the algorithm finds the **glb** of that quorum and then **lub** the result to the old state of the learner. If there are no compatible quorums left, then the collision recovery mechanism is initiated, which works as described in (LAMPORT et al., 2005):

> Learners also desirably detect if the acceptors have voted in such a way that no quorum is possible, and if so, the learner desirably prompts a leader to resolve the disagreement by starting a new ballot.

The issue that permeates Algorithm 3.7 lies in the high complexity operations that take form in the calculation of **glbs** and **lubs** to each possible quorum every time a new message arrives, which is a pretty naive approach to the problem. We discuss this in depth in Chapter 4 and propose an optimization.

Regarding the collision recovery, the approach used spends an extra step of communication when compared to the one proposed in (LAMPORT, 2004). This is so, because instead of detecting the collision at the learners it detects at the leader, by altering the protocol to include the leader as a receiver of messages 2B sent by the acceptors. The advantage of our approach is that although it adds an extra step of communication it is much simpler to implement, considering that in the other technique one would have to alter the leader to act as a learner to detect incompatible quorums. A deeper study of collision recovery can be found in (SUTRA; SHAPIRO, 2011).

3.2.10 Practical Considerations

Since the main goal of our project was not to create a high throughput and low latency implementation, but a faithful implementation of GPaxos protocol, our testing also went in this direction. Each phase of the protocol was locally tested through a set of hand generated messages designed with the aim to reach protocol corner cases. Towards the ending of the project, we tested the overall correctness by instantiating the protocol with

the different *CStruct* sets implemented, and sending randomly generated commands to a varying number of replicas (typically 5, but this number can be changed through the configuration files). By running an interactive version of the project instantiated with a chosen *CStruct* set, one can send commands to the replicas and attest that each one of them actually learns a compatible sequence of commands.

The project already has all the necessary structure needed for more complex testing, but before doing any stress related tests, our implementation would have to be strengthened in a series of ways. The first one would be the optimization of the *CStruct* set implementations, more specifically the CHistory *CStruct* set. Finding the **lub** and **glb** of CHistory c-structs implemented as graphs, is just too computationally complex. The second need lies in changing the naive learning algorithm seen in Section 3.2.9.1 to an optimized version, such as the one we provide in Chapter 4 that could be applied. Finally, a checkpointing mechanism is needed since the Generalized Paxos algorithm uses c-structs throughout, saving their values in variables and sending them in messages, an obvious problem when c-structs get too large. In Section 3.2.10.1 we briefly explain a checkpointing mechanism that could be implemented in our project.

3.2.10.1 Horizontal Checkpointing

Generalized Paxos continuously extends c-structs, possibly to contain the whole system history. Since they must be saved in variables, possibly on disk, and sent in messages, handling such large c-structs may become prohibitively expensive. To solve this issue, Lamport suggests (LAMPORT, 2004) using checkpoints and multiple Generalized Consensus instances, similarly to what is done in conventional state machine replication: when the c-structs exchanged during an instance are too large, a checkpoint command is proposed and a higher instance is started.

Differently from Lamport, we argue that the *horizontal checkpointing* technique described in (SUTRA, 2010) should be used. The technique consists in running a single instance of Generalized Consensus and successively checkpointing it over time.

Horizontal checkpointing leverages the observation that once a c-struct is learned, it is by definition stable and, therefore, can be forgotten. Nonetheless, as failures can occur, discarding part of a c-struct should be done with care.

In a nutshell, the technique achieves this as follows: When a process appends a checkpoint command to some c-struct, it trims the c-struct up to the previous checkpoint command. Two c-structs are comparable (using relation $\sqsubseteq$) if they suffix the same checkpoint.

There are several advantages in using horizontal checkpointing over multiple instances of Generalized Paxos. First of all, the approach fully embraces the genericity brought by the c-struct abstraction, requiring a single communication stack to solve generalized consensus. Second, as horizontal checkpointing uses a single instance of Generalized Paxos,

the technique reduces the memory footprint. Third, multiple instances have to run in sequence one after the other, which requires a stop-the-world mechanism, where a checkpoint command operation is decided then a new instance of the protocol started. The Horizontal Checkpointing technique solely requires to decide upon the checkpoint command, it saves two message delay. New instances may still serve during a reconfiguration, which are much more rare events.

3.2.10.2 On Testing the Protocol

A reasonable test of the protocol should consider different metrics, such latency, throughput, number of rounds started per message, etc, under different workloads. Furthermore, tests must multiple *CStruct* sets as well as different rates of commuting commands. These necessities can already be altered in our implementation, for example, the *commuteWith* method from the *Commutative Command Int* class explained in Section 3.2.3, could be easily changed to analyze the impact of the amount of commuting commands in the system. For example, suppose one wants to issue N random commands to the system and wants that deterministically 25% of them to commute. To achieve this task one could set up an integer limit L for the *content* variable and when implementing the *commuteWith* method, force that only the integers bounded by an interval of $|L/4|$ commute between themselves (e.g. $L = 1000$, integers from 1 to 250 commute). Then, one could populate a set with 25% of N with the random integers in the chosen interval of the *commuteWith* method and the rest with random integers not in the interval.

We also argue that a reasonable testing environment for the GPaxos protocol should consider a data center topology, that consists in groups of replicas with low latency between themselves and high latency to other groups of replicas. In this kind of topology, collisions shall be more frequent, allowing one to check how efficient is GPaxos protocol when handling them.

Thankfully to the framework used and the tools on which it was build upon (i.e. Akka and Scala), deploying our implementation in such a topology is just a matter of altering configuration files, a process that can be automated through simple scripts.

Chapter 4

Theoretical Contribution

4.1 A *CStruct* set for distributed lease coordination

The different definitions of *CStruct* sets allow solving a variety of problems using a single and highly optimized GPaxos implementation. Hence, new *CStruct* sets, for different problems, are badly needed in order to use such a potential in practical systems.

Large-scale distributed systems often require scalable and fault-tolerant mechanisms to coordinate exclusive access to shared resources such as files (or disk blocks, see Figure 12).

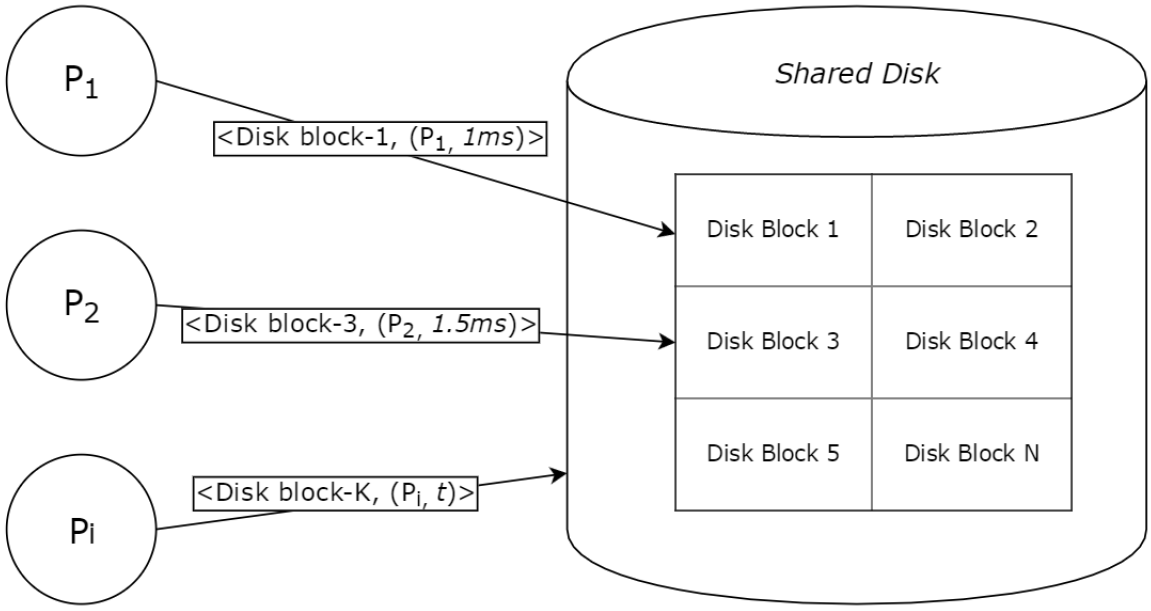

Figure 12 – Multiple processes p_i trying to acquire access for time t (lease) to the shared resource.

A lease is a token which grants its owner exclusive access to a resource for a defined span of time. In order to tolerate failures, leases must be coordinated by distributed processes. The best-known algorithms to implement distributed mutual exclusion with leases, such as Multipaxos, are complex and difficult to implement(KOLBECK et al., 2011); the Chubby lock service(BURROWS, 2006) is an example of such a system.

We now present a novel *CStruct* set, which allows GPaxos to solve a variation of the distributed lease coordination problem by encapsulating its semantics into the append operation. The solution is surprisingly simple and easy to implement. As of our current knowledge, this is the first time in which the Generalized Paxos algorithm is used to solve a variation of the lease coordination problem in a distributed system and one of the few applications of Generalized Consensus out of the set initially documented by Lamport(LAMPORT, 2004).

4.1.1 Definitions

We now extend our system model to assume that each process p_i has access to a local (hardware) clock c_i, similarly to (CRISTIAN; FETZER, 1999). Clocks are loosely synchronized. This means that the clock drift between any two processes is less than or equal to ϵ. To guarantee progress, we assume that the maximum time span of lease validity is T_{max}, and that $T_{max} > \epsilon$ (KOLBECK et al., 2011). We also assume that no lease is smaller than ϵ.

When a process needs access to a critical section, it sends a message to the proposers containing a *lease* command with the following structure:

$$C = \langle CS_k, Process, T_{begin}, T_{end} \rangle$$

The first value of the tuple identifies the critical section, the second value identifies the process, and, the last two, the beginning and end of the lease, respectively.

We expect that the lease issued by the process represents an immediate need, but that it could be postponed for a later time. In particular, if the critical section is already in use, the process might wait until after the last request in the waiting queue has been serviced. The lease duration is, however, not altered. This semantics of a lease request is the most common, and it guarantees that each process eventually gains access the critical section for the duration of its original request (i.e., no starvation).

We now define a set of c-structs, denoted $CLease$, that is well-suited for the lease coordination problem. Each c-struct in $CLease$ is a map where the keys are the critical section identifiers and the values are queues of *lease* commands requesting leases to such critical sections, as described above in this section. The empty c-struct, $\perp$, is the empty map. In order to append a command C to a c-struct $LeaseMap$, a protocol agent executes Algorithm 4.1. To simplify the presentation we treat a missing key as one with an empty queue, in the algorithm.
If $f : X \to Y$ is a function (or mapping) from domain X to range Y, then $Domain(f) = X$ and $Range(f) = Y$, then we can describe $CLease$'s properties as follows:

1. Algorithm 4.1 uniquely defines the append operation $\bullet$. Considering two sequences of *lease* commands $\sigma \sim \tau$ and the *LeaseMaps*: $u = \perp \bullet \sigma$ and $v = \perp \bullet \tau$, the append

Algorithm 4.1 *CLease* append algorithm

```
1: Task LeaseMap • C
2:     ∧ϵ                                                    // pre-conditions
3:     ∧t_max > ϵ
4:     if LeaseMap[C.CS] = ∅ then
5:         nLease := ⟨C.Process, C.T_begin, C.T_end⟩
6:     else
7:         last_elem := LeaseMap[C.CS].last()
8:         nTbegin := last_elem.T_end + ϵ
9:         nT := C.T_end − C.T_begin
10:        nTend := nTbegin + nT
11:        nLease := ⟨C.Process, nTbegin, nTend⟩
12:    LeaseMap[C.CS].enqueue(nLease)
```

algorithm in this two c-structs will yield two c-structs $u, v \in CLease$, such that $u = v$. These two c-structs are equal because the keys and the values they map for are equal. In a more mathematical way: $\forall u, v \in CLease,\ u = v \Rightarrow (Domain(u) = Domain(v)) \wedge (Range(u) = Range(v))$

2. Given c-structs $C, D \in CLease$, C is prefix of D ($C \sqsubseteq D$) iff the set comprehending of the intersection of keys between both c-structs is non-empty and, for each key in this intersection, the mapped queue in C is a prefix of the mapped queue in D. More formally: $\forall u, v \in CLease,\ u \sqsubseteq v \Rightarrow I = Domain(u) \cap Domain(v) \neq \emptyset \wedge \forall k \in I, v[k] \sqsubseteq u[k]$.

3. Two c-structs in $CLease$ are compatible iff the set comprehending of the intersection of keys between both c-structs is empty, or if the set is non-empty, then for each such key, the queues it maps to are compatible; that is, they share an upper bound.

It is possible to observe that the axioms in 2.5.2 are met by the properties in the following way: CS1 is met by property 1, CS2 by property 2 and CS3 and CS4 by property 3.

Upon learning a command with its identifier, the client is free to access the resource within the respective time boundaries, a period for which we say the lease is **ongoing**.

4.1.2 The case for Generalized Consensus

It is commonly believed that Generic Broadcast (PEDONE; SCHIPER, 2002) and Generalized Consensus (GBCast) are similar algorithms. In particular, Lamport (LAMPORT,) writes that

> the difference in efficiency between the two algorithms is insignificant.

As a consequence, one might conclude that performance gains of Generalized Consensus stem from instantiating c-structs as command histories, in which case GBCast would work just as well. We claim here that this is not accurate because while Generic Broadcast looks only at pairwise conflict relations, the append operator of a c-struct set can look at the set of accepted commands before appending a new one. For example, one could devise a k-mutual exclusion (RAYMOND, 1989) algorithm based on $CLease$, in which a request is accepted only if it will not cause more than k leases to be ongoing at the same time, which is not possible with GBCast.

4.2 An Optimization To Generalized Paxos

In (LAMPORT, 2004), Lamport explores the expressiveness of the TLA+ to provide a concise specification for Generalized Paxos. Naively translating such specification would result in inefficient code, in that it may, for example, solve harder problems than actually needed or always compute from scratch instead of incrementally from previous results. In 3.2.8.1, we presented an optimization implemented in our GPaxos protocol, that allows for faster starting a new ballot. In a similar fashion, the next section presents our proposal for reaching an agreement with a less computationally expensive algorithm.

4.2.1 Learner's optimized quorum detection

The computation of **lubs** and **glbs** is the most expensive part of GPaxos. In what follows, we show that in particular during the learning phase this cost is high and we develop a solution to decrease it.

4.2.1.1 An expensive computation

When a 2B message from some acceptor a is received at process p_i (lines 58 to 60 in Algorithm 2.1), a straightforward implementation of the *learn* action goes as follows:

> Find the set $\mathcal{Q}$ of all the possible quorums of acceptors Q with $a \in Q$ and then check if the c-structs accepted by $Q \in \mathcal{Q}$ are compatible. If this is the case, calculate the glb of these c-structs to find the greatest common prefix u, then assign $\sqcup \{learned_i, u\}$ to $learned_i$.

Hence, we need to calculate a glb followed by a lub for each quorum including a. The cost of this computation depends on the definition of *CStruct*. Considering that we have m c-structs with n elements each, for example, the cost is $O(nm)$ when c-structs are CSeqs and $O(n^2m)$ when c-histories. [1] Hence, in the worth case, the complexity of the learning phase belongs $O(qn^2m)$, where q is the number of possible quorums, m.

[1] A c-history is directed acyclic graph, and it takes $O(n)$ operations to append a new command to a graph already containing n nodes.

Each time a new 2B message is received, a new quorum may be formed that extends the learned c-struct even further. As the above computation takes place very often, it becomes a burden for the algorithm and may erase all the performance gains brought by Generalized Consensus.

4.2.1.2 A Trie-like solution

To improve the learning phase of GPaxos, our solution is to store the accepted c-structs in a Trie-like data structure (or c-trie, for short) that grows over time. In detail, our technique works as follows:

- When an acceptor forwards the c-struct u it accepted, it sends in addition a command sequence σ such that $u = \bot \bullet \sigma$.
- A learner that receives a c-struct with its respective σ from some acceptor a executes Algorithm 4.2 to update its c-trie and possibly detect that an agreement occurs.
- In Algorithm 4.2, the learner maintains the root of the c-trie (*root*) and a hash map ($find$) binding the c-structs to the nodes of the c-trie. That map provides constant time access to the prefixes. It is built over time with the help of a (perfect) hash function denoted *hash*. Each node of the c-trie maintains its children (field *child*), a c-struct (field *value*), and the number of acceptors that voted for that c-struct (field *votes*).

For illustration purposes, consider the Chistory $CStruct$ set and 3 commands A, B and C, such that A commutes with C, and B conflicts with both A and C ($B \asymp A$ and $B \asymp C$). Assume that the acceptors are on a fast ballot and that solely A, B and C are proposed.

As the learner can receive the messages containing the votes from the acceptors (i.e. c-structs) in an unordered manner, it can receive for example a c-struct containing 3 commands before receiving the same c-struct with 1 command. Suppose a fast ballot with 8 acceptors ($QSize = 6$) and that a learner received unordered votes from these acceptors. Initially the 8 acceptors voted for the c-struct $e = \bot$, later on some collisions happened causing 4 acceptors to vote for c-struct $u_1 = \bot \bullet A$ and the others to vote for c-struct $u_2 = \bot \bullet C$. New votes arrive revealing that from the 8 acceptors that voted for c-structs u_1 and u_2, 3 of them received the command C and 3 received command A, which allows the algorithm to register 6 votes for the c-trie node that contains c-struct $v = \bot \bullet A \bullet C$, since $u_1 \sqsubseteq v$ and $u_2 \sqsubseteq v$. The remaining 2 acceptors received the interfering command B after receiving commands A or C, which makes them vote for the c-structs $x = \bot \bullet A \bullet B$ and $y = \bot \bullet C \bullet B$ that are not $\sqsubseteq$ of v. The insertion algorithm then registers 1 vote for each c-struct in different nodes of the c-trie. In this scenario only the

c-structs e and v received an amount of votes greater or equal to the quorum size, after searching the c-trie for a quorum e will be learned followed by its extension, v.

Figure 13 details the final state of the c-trie during the aforementioned scenario.

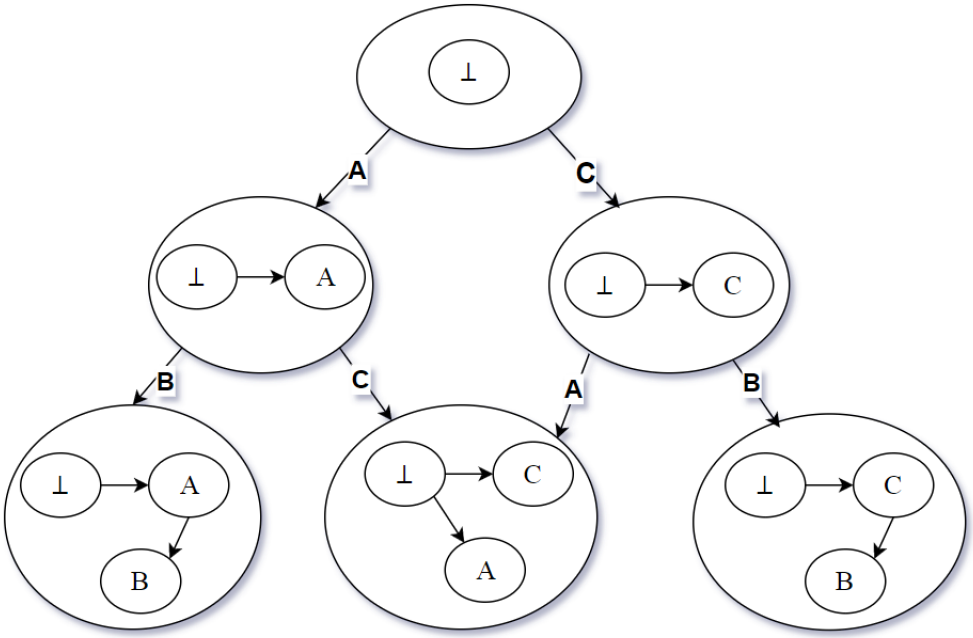

Figure 13 – Trie-like structure used for quorum detection

Consider that n is the length of the command sequence σ. In the worst case, the insertion of a c-struct in the c-trie requires $O(n^2)$ operation (due to line 11 in Algorithm 4.2). Moreover, finding a c-struct that is accepted by a quorum takes $O(dm)$ time, where d is the depth of the c-trie and m the total number of acceptors (each level of the c-trie can have at most $|Acceptors|$). Hence, in total, we execute $O(n^2 + dm)$ operations instead of $O(qn^2m)$ in the initial solution.

Notice that once we know that a c-struct prefixes all the c-structs accepted by the acceptors (typically at the start of a new ballot), we can prune the tree and make this c-struct the new root of the c-trie.

Algorithm 4.2 Trie-like insertion algorithm

```
1:  Task insert(σ, a)
2:      node := root                                   // root of the c-trie
3:      i := 0
4:      n := length(σ)
5:      while i < n do
6:          if node.child(σ[i]) ≠ nil then
7:              node.votes := node.votes ∪ {a}
8:              node := node.child(σ[i])
9:          else
10:             let u = node.value
11:             v := u • σ[i]
12:             n := find(v)                           // query hash map
13:             if n ≠ nil then
14:                 node.child(σ[i]) := n
15:                 node := n
16:                 node.votes := node.votes ∪ {a}
17:             else
18:                 node.child(σ[i]) = new node
19:                 find := find ∪ {hash(v), node}
20:                 node.value = v
21:                 node := node.child(σ[i])
22:         i := i + 1
```

Chapter 5

Conclusion

Generalized Paxos offers an elegant and efficient solution to multiple distributed problems. However, to date, few implementations exist as the algorithm is difficult to understand and its applicability not well understood. This work steps towards reversing this situation by, first, presenting a pseudo-code description that complements the original one, given by (LAMPORT, 2004), as well as a complete overview of the Generalized Consensus problem and related works.

Second, we provide an open source implementation that closely follows Lamport's TLA+ and that one can easily trace back all the actions from the specification to code in order to check its correctness. The implementation used in (MORARU; ANDERSEN; KAMINSKY, 2013) of the Generalized Paxos protocol, has no different *CStruct* sets that one can instantiate the algorithm with, thus is not faithfully enough to the specification. The implementation in (SUTRA, 2010) is outdated technologically in comparison to ours, making it harder to extend and test. It also assumes a single quorum per ballot, which should be an optional mode of execution instead of the only possible one.

Finally, we introduce a novel *CStruct* set for a variation of the lease coordination problem that underlines the versatility of the Generalized Paxos algorithm as well as an optimization regarding the detection of an agreement; with this work we are closer to achieving an Agreement-as-a-Service infrastructure than before.

5.1 Future Work

Primarily we aim at fulfilling the issues discussed in Section 3.2.10, so that the protocol can be tested in more challenging situations, such as emulating a geo-replicated datacenter with a realistic number of replicas. After the performance related changes, it is important to provide stable-storage to the acceptor agents in order to have a *Crash-recovery* implementation.

Furthermore, a proof-of-concept implementation for the *CLease CStruct* set presented in Chapter 4, would heavily endorse the claims presented in this work. Since our imple-

mentation is easily extensible, implementing the multicoordinated ballots proposed in (CAMARGOS; SCHMIDT; PEDONE, 2007) could yield results that attest the benefits of the technique.

Bibliography

Akka team. **Membership Lifecycle**. 2016. <http://doc.akka.io/docs/akka/2.4.10/common/cluster.html#Membership_Lifecycle>. Accessed: 2016-12-19.

BEN-OR, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: **Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 1983. (PODC '83), p. 27–30. ISBN 0-89791-110-5. Disponível em: <http://doi.acm.org/10.1145/800221.806707>.

BERNSTEIN, P.; HADZILACOS, V.; GOODMAN, N. **Concurrency Control and Recovery in Database Systems**. Addison-Wesley, 1987. Disponível em: <http://research.microsoft.com/pubs/ccontrol/>.

BRACHA, G.; TOUEG, S. Resilient consensus protocols. In: **Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 1983. (PODC '83), p. 12–26. ISBN 0-89791-110-5. Disponível em: <http://doi.acm.org/10.1145/800221.806706>.

BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In: **Proc. of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2006. p. 24–24. Disponível em: <http://portal.acm.org/citation.cfm?id=1267308.1267332>.

BUTCHER, P. **Seven Concurrency Models in Seven Weeks: When Threads Unravel**. 1st. ed. [S.l.]: Pragmatic Bookshelf, 2014. ISBN 1937785653, 9781937785659.

CAMARGOS, L.; SCHMIDT, R.; PEDONE, F. **Multicoordinated Paxos**. [S.l.], 2006.

CAMARGOS, L. J. **Multicoordinated agreement protocols and the log service**. Tese (Doutorado) — Università della Svizzera italiana, 4 2008.

CAMARGOS, L. J.; SCHMIDT, R. M.; PEDONE, F. Multicoordinated paxos. In: **Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2007. (PODC '07), p. 316–317. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281150>.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: An engineering perspective. In: **Proc. of the 26th Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2007. (PODC '07), p. 398–407. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281103>.

CHANDRA, T. D. et al. On the impossibility of group membership. In: **Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 1996. (PODC '96), p. 322–330. ISBN 0-89791-800-2. Disponível em: <http://doi.acm.org/10.1145/248052.248120>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Communications of the ACM**, v. 43, n. 2, p. 225–267, 1996. Disponível em: <http://www.acm.org/pubs/toc/Abstracts/jacm/226647.html>.

CRISTIAN, F. Reaching agreement on processor group membership in synchronous distributed systems. **Distributed Computing**, v. 4, p. 175–187, 1991.

CRISTIAN, F.; FETZER, C. The timed asynchronous distributed system model. **IEEE Transactions on Parallel and Distributed Systems**, IEEE Computer Society, Washington, DC, USA, p. 642–657, June 1999. ISSN 1045-9219.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. In: **IN PROC. SOSP**. [S.l.: s.n.], 2007. p. 205–220.

DéFAGO, X. et al. The phi accrual failure detector. In: **RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology**. [S.l.: s.n.], 2004. p. 66–78.

EUGSTER, P. T. et al. Epidemic information dissemination in distributed systems. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 5, p. 60–67, maio 2004. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2004.1297243>.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **J. ACM**, ACM Press, v. 32, n. 2, p. 374–382, April 1985. ISSN 0004-5411. Disponível em: <http://portal.acm.org/citation.cfm?id=214121>.

HADZILACOS, V.; TOUEG, S. **A Modular Approach to Fault-Tolerant Broadcasts and Related Problems**. [S.l.], 1994.

HEWITT, C.; BISHOP, P.; STEIGER, R. A universal modular actor formalism for artificial intelligence. In: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. (IJCAI'73), p. 235–245. Disponível em: <http://dl.acm.org/citation.cfm?id=1624775.1624804>.

KIRSCH, J.; AMIR, Y. Paxos for system builders: An overview. In: **Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware**. New York, NY, USA: ACM, 2008. (LADIS '08), p. 3:1–3:6. ISBN 978-1-60558-296-2. Disponível em: <http://doi.acm.org/10.1145/1529974.1529979>.

KOLBECK, B. et al. Flease - lease coordination without a lock server. In: **Proc. of the 2011 IEEE International Parallel & Distributed Processing Symposium**. Washington, DC, USA: IEEE Computer Society, 2011. (IPDPS '11), p. 978–988. ISBN 978-0-7695-4385-7. Disponível em: <http://dx.doi.org/10.1109/IPDPS.2011.94>.

LAMPORT, L. **The Writings of Leslie Lamport**. <http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html>. Accessed: 2016-10-26.

_____. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, ACM Press, New York, NY, USA, v. 21, n. 7, p. 558–565, July 1978. ISSN 0001-0782. Disponível em: <http://portal.acm.org/citation.cfm?id=359563>.

_____. The part-time parliament. **ACM Trans. Comput. Syst.**, ACM Press, v. 16, n. 2, p. 133–169, May 1998. ISSN 0734-2071. Disponível em: <http://portal.acm.org/citation.cfm?id=279227.279229>.

_____. Paxos made simple. **ACM SIGACT News (Distributed Computing Column)**, v. 32, n. 4, p. 18–25, December 2001.

_____. **Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers**. Addison-Wesley Professional, 2002. Paperback. ISBN 032114306X. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{&}path=ASIN/0321143>.

_____. **Generalized Consensus and Paxos**. [S.l.], 2004.

_____. Fast paxos. **Distributed Computing**, v. 19, n. 2, p. 79–103, 2006. ISSN 1432-0452. Disponível em: <http://dx.doi.org/10.1007/s00446-006-0005-x>.

_____. Lower bounds for asynchronous consensus. **Distributed Computing**, Springer-Verlag, London, UK, v. 19, n. 2, p. 104–125, October 2006.

Leslie Lamport, Jonathan Howell, Jacob Lorch e John Douceur. **Automatic commutativity detection for generalized paxos**. 2005. US 20060184627A1. Disponível em: <https://patents.google.com/patent/US20060184627A1/>.

MAO, Y.; JUNQUEIRA, F. P.; MARZULLO, K. Mencius: building efficient replicated state machines for wans. In: **Proc. of the 8th USENIX conference on Operating systems design and implementation**. Berkeley, CA, USA: USENIX Association, 2008. p. 369–384.

MAZURKIEWICZ, A. Advances in petri nets 1984. In: ROZENBERG, G. (Ed.). London, UK, UK: Springer-Verlag, 1985. cap. Semantics of Concurrent Systems: A Modular Fixed-point Trace Approach, p. 353–375. ISBN 0-387-15204-0. Disponível em: <http://dl.acm.org/citation.cfm?id=28461.28480>.

MORARU, I.; ANDERSEN, D. G.; KAMINSKY, M. There is more consensus in egalitarian parliaments. In: **Proc. of the 24th ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. (SOSP '13), p. 358–372. ISBN 978-1-4503-2388-8. Disponível em: <http://doi.acm.org/10.1145/2517349.2517350>.

ODERSKY, M.; SPOON, L.; VENNERS, B. **Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition**. 2nd. ed. USA: Artima Incorporation, 2011. ISBN 0981531644, 9780981531649.

PEDONE, F.; SCHIPER, A. Handling message semantics with generic broadcast protocols. **Distributed Computing**, Springer-Verlag, London, UK, v. 15, n. 2, p. 97–107, April 2002. ISSN 0178-2770. Disponível em: <http://dx.doi.org/10.1007/s004460100061>.

RAYMOND, K. A distributed algorithm for multiple entries to a critical section. **Information Processing Letters**, v. 30, n. 4, p. 189 – 193, 1989. ISSN 0020-0190. Disponível em: <http://www.sciencedirect.com/science/article/pii/0020019089902111>.

RAYNAL, M.; SINGHAL, M. Mastering agreement problems in distributed systems. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 18, n. 4, p. 40–47, jul. 2001. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/MS.2001.936216>.

SARAMAGO, R. Q. **Implementação e Avaliação do Protocolo de Difusão Atômica Rápida à Despeito de Colisões**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2016.

SCHMIDT, R.; CAMARGOS, L.; PEDONE, F. Collision-fast atomic broadcast. In: **Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications**. Washington, DC, USA: IEEE Computer Society, 2014. (AINA '14), p. 1065–1072. ISBN 978-1-4799-3630-4.

SUBRAMANIYAN, R. et al. Gems: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. **Cluster Comput**, 2006.

SUTRA, P. **Efficient Protocols for Generalized Consensus and Partial Replication**. Tese (Doutorado) — Université Pierre et Marie CURIE, 11 2010.

SUTRA, P.; SHAPIRO, M. Fast Genuine Generalized Consensus. Working paper or preprint. 2010. Disponível em: <https://hal.archives-ouvertes.fr/hal-00476885>.

_____. Fast genuine generalized consensus. In: **Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on**. [S.l.: s.n.], 2011. p. 255–264. ISSN 1060-9857.

TURCU, A. et al. Be general and don't give up consistency in geo-replicated transactional systems. In: **Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings**. [s.n.], 2014. p. 33–48. Disponível em: <http://dx.doi.org/10.1007/978-3-319-14472-6_3>.

VAZIRI, M. et al. Declarative object identity using relation types. In: **Proceedings of the 21st European Conference on Object-Oriented Programming**. Berlin, Heidelberg: Springer-Verlag, 2007. (ECOOP'07), p. 54–78. ISBN 3-540-73588-7, 978-3-540-73588-5. Disponível em: <http://dl.acm.org/citation.cfm?id=2394758.2394764>.

VIEIRA, G. M.; BUZATO, L. E. On the coordinator's rule for fast paxos. **Information Processing Letters**, v. 107, n. 5, p. 183–187, 2008.

VIEIRA, G. M. D.; BUZATO, L. E. The performance of paxos and fast paxos. **CoRR**, abs/1308.1358, 2013. Disponível em: <http://arxiv.org/abs/1308.1358>.

ZHAO, W. Fast paxos made easy: Theory and implementation. **Int. J. Distrib. Syst. Technol.**, IGI Global, Hershey, PA, USA, v. 6, n. 1, p. 15–33, jan. 2015. ISSN 1947-3532. Disponível em: <http://dx.doi.org/10.4018/ijdst.2015010102>.

ZIELINSKI, P. Optimistic generic broadcast. In: **Proc. of the 19th International Symposium on Distributed Computing**. Kraków, Poland: [s.n.], 2005. p. 369–383.

Annex

ANNEX A

Generalized Paxos TLA+

C.6 The Generalized Paxos Consensus Algorithm

MODULE *GeneralizedPaxos*

EXTENDS *PaxosConstants*

We introduce a set *Leader* of leaders, and let $LeaderOf(m)$ be the leader of ballot number m.

CONSTANT $Leader$, $LeaderOf(\_)$

ASSUME $\forall\, m \in BalNum : LeaderOf(m) \in Leader$

The set *Msg* of all possible messages is the same as for the distributed abstract algorithm of module *DistAbstractGPaxos*.

$$
\begin{array}{lll}
Msg \triangleq & [type : \{\text{“propose”}\},\ cmd : Cmd] \\
\cup & [type : \{\text{“1a”}\},\ bal : BalNum] \\
\cup & [type : \{\text{“1b”}\},\ bal : BalNum,\ acc : Acceptor, \\
 & \quad vbal : BalNum,\ vote : CStruct \cup \{none\}] \\
\cup & [type : \{\text{“2a”}\},\ bal : BalNum,\ val : CStruct] \\
\cup & [type : \{\text{“2b”}\},\ bal : BalNum,\ acc : Acceptor,\ val : CStruct]
\end{array}
$$

We define *NotABalNum* to be an arbitrary value that is not a balnum.

$NotABalNum \triangleq$ CHOOSE $m : m \notin BalNum$

The variables *propCmd*, *learned*, and *msgs* are the same as in the distributed abstract algorithm. We replace the abstract algorithm's variable *bA* with the variables *mbal*, *bal*, and *val*. The value of $curLdrBal[ldr]$ is the ballot that leader *ldr* is currently leading or has most recently led. Initially, its value is initially $curLdrBal[ldr]$ equals *NotABalNum* for all leaders except the leader of ballot 0, which is initially in progress. We replace the variable *maxTried* of the abstract algorithm with *maxLdrTried*, where the value of $maxLdrTried[ldr]$ corresponds to the value of $maxTried[curLdrBal[ldr]]$ in the abstract algorithm.

VARIABLES $propCmd$, $learned$, $msgs$, $maxLdrTried$, $curLdrBal$, $mbal$, bal, val

We begin with the type invariant and the initial predicate.

$$
\begin{aligned}
TypeInv \triangleq\ & \land\ propCmd \subseteq Cmd \\
& \land\ learned \in [Learner \to CStruct] \\
& \land\ msgs \subseteq Msg \\
& \land\ maxLdrTried \in [Leader \to CStruct \cup \{none\}] \\
& \land\ curLdrBal \in [Leader \to BalNum \cup \{NotABalNum\}] \\
& \land\ mbal \in [Acceptor \to BalNum] \\
& \land\ bal \in [Acceptor \to BalNum] \\
& \land\ val \in [Acceptor \to CStruct]
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land\ propCmd = \{\} \\
& \land\ learned = [l \in Learner \mapsto Bottom] \\
& \land\ msgs = \{\} \\
& \land\ maxLdrTried = [ldr \in Leader \mapsto \text{IF } ldr = LeaderOf(0) \\
& \qquad\qquad\qquad\qquad \text{THEN } Bottom \text{ ELSE } none] \\
& \land\ curLdrBal = [ldr \in Leader \mapsto \text{IF } ldr = LeaderOf(0) \\
& \qquad\qquad\qquad\qquad \text{THEN } 0 \text{ ELSE } NotABalNum] \\
& \land\ mbal = [a \in Acceptor \mapsto 0] \\
& \land\ bal = [a \in Acceptor \mapsto 0] \\
& \land\ val = [a \in Acceptor \mapsto Bottom]
\end{aligned}
$$

We now define the actions.

$$
\begin{aligned}
& SendProposal(C) \triangleq \\
& \quad \land\ propCmd' = propCmd \cup \{C\} \\
& \quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“propose”},\ cmd \mapsto C]\} \\
& \quad \land\ \text{UNCHANGED } \langle learned,\ maxLdrTried,\ curLdrBal,\ mbal,\ bal,\ val\rangle
\end{aligned}
$$

$$
\begin{aligned}
& Phase1a(ldr) \triangleq \\
& \quad \land\ \exists\, m \in BalNum : \\
& \qquad \land\ \lor\ curLdrBal[ldr] = NotABalNum \\
& \qquad\quad \lor\ curLdrBal[ldr] < m \\
& \qquad \land\ LeaderOf(m) = ldr
\end{aligned}
$$

$\land\ curLdrBal' = [curLdrBal \text{ EXCEPT } ![ldr] = m]$
$\land\ maxLdrTried' = [maxLdrTried \text{ EXCEPT } ![ldr] = none]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“1a”},\ bal \mapsto m]\}$
$\land$ UNCHANGED $\langle propCmd,\ learned,\ mbal,\ bal,\ val\rangle$

$Phase1b(a,\ m) \triangleq$
$\land\ [type \mapsto \text{“1a”},\ bal \mapsto m] \in msgs$
$\land\ mbal[a] < m$
$\land\ mbal' = [mbal \text{ EXCEPT } ![a] = m]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“1b”},\ bal \mapsto m,\ acc \mapsto a,$
$vbal \mapsto bal[a],\ vote \mapsto val[a]]\}$
$\land$ UNCHANGED $\langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ bal,\ val\rangle$

$Phase2Start(ldr,\ v) \triangleq$
$\land\ curLdrBal[ldr] \neq NotABalNum$
$\land\ maxLdrTried[ldr] = none$
$\land\ \exists\, Q \in Quorum(curLdrBal[ldr]) :$
$\land\ \forall\, a \in Q : \exists\, msg \in msgs :\ \land\ msg.type = \text{“1b”}$
$\land\ msg.bal = curLdrBal[ldr]$
$\land\ msg.acc = a$
$\land$ LET We define *PrSafe* so it equals the value of $ProvedSafe(Q, m, beta)$ where computed in the corresponding action of the distributed abstract algorithm. To help understand this correspondence, see the definition of *ProvedSafe* in module *PaxosConstants*.
$1bMsg(a) \triangleq$
For an acceptor a in Q, this is the “1b” message sent by a for ballot number $curLdrBal[ldr]$. There can be only one such message.
CHOOSE $msg \in msgs :\ \land\ msg.type = \text{“1b”}$
$\land\ msg.bal = curLdrBal[ldr]$
$\land\ msg.acc = a$
$k \triangleq Max(\{1bMsg(a).vbal : a \in Q\})$
$RS \triangleq \{R \in Quorum(k) : \forall\, a \in Q \cap R : 1bMsg(a).vbal = k\}$
$g(R) \triangleq GLB(\{1bMsg(a).vote : a \in Q \cap R\})$
$G \triangleq \{g(R) : R \in RS\}$
$PrSafe \triangleq$
When the action is enabled, the set G will always be compatible.
IF $RS = \{\}$ THEN $\{1bMsg(a).vote :$
$a \in \{b \in Q : 1bMsg(b).vbal = k\}\}$
ELSE $\{LUB(G)\}$
$pCmd \triangleq \{msg.cmd : msg \in \{mg \in msgs : mg.type = \text{“propose”}\}\}$
IN $\land\ \exists\, w \in PrSafe,\ s \in Seq(pCmd) :$

$$\begin{aligned}
&\land\ maxLdrTried' = [maxLdrTried \text{ EXCEPT } ![ldr] = w *\!* s] \\
&\land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto curLdrBal[ldr],\\
&\qquad\qquad\qquad\qquad val\ \mapsto w *\!* s]\} \\
&\land \text{UNCHANGED } \langle propCmd,\ learned,\ curLdrBal,\ mbal,\ bal,\ val\rangle
\end{aligned}$$

$$\begin{aligned}
&Phase2aClassic(ldr,\ C)\ \triangleq \\
&\land\ curLdrBal[ldr] \neq NotABalNum \\
&\land\ [type \mapsto \text{“propose”},\ cmd \mapsto C] \in msgs \\
&\land\ maxLdrTried[ldr] \neq none \\
&\land\ maxLdrTried' = [maxLdrTried \text{ EXCEPT } ![ldr] = maxLdrTried[ldr] \bullet C] \\
&\land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto curLdrBal[ldr],\\
&\qquad\qquad\qquad\qquad val \mapsto maxLdrTried'[ldr]]\} \\
&\land \text{UNCHANGED } \langle propCmd,\ learned,\ curLdrBal,\ mbal,\ bal,\ val\rangle
\end{aligned}$$

$$\begin{aligned}
&Phase2bClassic(a,\ v)\ \triangleq \\
&\land\ [type \mapsto \text{“2a”},\ bal \mapsto mbal[a],\ val \mapsto v] \in msgs \\
&\land\ \lor\ bal[a] < mbal[a] \\
&\quad\ \lor\ val[a] \sqsubset v \\
&\land\ bal' \ \ = [bal \text{ EXCEPT } ![a] = mbal[a]] \\
&\land\ val' \ \ = [val \text{ EXCEPT } ![a] = v] \\
&\land\ msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto mbal[a],\ acc \mapsto a,\ val \mapsto v]\} \\
&\land \text{UNCHANGED } \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ mbal\rangle
\end{aligned}$$

$$\begin{aligned}
&Phase2bFast(a,\ C)\ \triangleq \\
&\land\ [type \mapsto \text{“propose”},\ cmd \mapsto C] \in msgs \\
&\land\ bal[a] = mbal[a] \\
&\land\ val' = [val \text{ EXCEPT } ![a] = val[a] \bullet C] \\
&\land\ msgs' = msgs \cup \\
&\qquad\qquad \{[type \mapsto \text{“2b”},\ bal \mapsto mbal[a],\ acc \mapsto a,\ val \mapsto val'[a]]\} \\
&\land \text{UNCHANGED } \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ mbal,\ bal\rangle
\end{aligned}$$

$$\begin{aligned}
&Learn(l,\ v)\ \triangleq \\
&\land\ \ \exists\, m \in BalNum : \\
&\qquad \exists\, Q \in Quorum(m) : \\
&\qquad\quad \forall\, a\ \ \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{“2b”} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.bal\ \ = m \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.acc\ \ = a \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ v \sqsubseteq msg.val \\
&\land\ \ learned' = [learned \text{ EXCEPT } ![l] = learned[l] \sqcup v] \\
&\land\ \ \text{UNCHANGED } \langle propCmd,\ msgs,\ maxLdrTried,\ curLdrBal,\ mbal,\ bal,\ val\rangle
\end{aligned}$$

Next and *Spec* are the complete next-state relation and specification.

$$
\begin{array}{lll}
\mathit{Next} \triangleq & \lor \exists\, C \in \mathit{Cmd} : \mathit{SendProposal}(C) & \text{The proposers' actions.} \\
& \lor \exists\, \mathit{ldr} \in \mathit{Leader} : & \text{The leaders' actions.} \\
& \quad \lor \mathit{Phase1a}(\mathit{ldr}) & \\
& \quad \lor \exists\, v \in \mathit{CStruct} : \mathit{Phase2Start}(\mathit{ldr},\, v) & \\
& \quad \lor \exists\, C \in \mathit{Cmd} : \quad \mathit{Phase2aClassic}(\mathit{ldr},\, C) & \\
& \lor \exists\, a \in \mathit{Acceptor} : & \text{The acceptors' actions.} \\
& \quad \lor \exists\, v \in \mathit{CStruct} : \mathit{Phase2bClassic}(a,\, v) & \\
& \quad \lor \exists\, m \in \mathit{BalNum} : \mathit{Phase1b}(a,\, m) & \\
& \quad \lor \exists\, C \in \mathit{Cmd} \quad : \mathit{Phase2bFast}(a,\, C) & \\
& \lor \exists\, l \in \mathit{Learner} : & \text{The learners' actions.} \\
& \quad \exists\, v \in \mathit{CStruct} : \mathit{Learn}(l,\, v) &
\end{array}
$$

$$
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{propCmd},\, \mathit{learned},\, \mathit{msgs},\, \mathit{maxLdrTried},\, \mathit{curLdrBal},\, \mathit{mbal},\, \mathit{bal},\, \mathit{val} \rangle}
$$

The following theorems assert that *TypeInv* is an invariant and that *Spec* implements the specification of generalized consensus, formula *Spec* of module *GeneralConsensus*.

THEOREM $\mathit{Spec} \Rightarrow \Box\, \mathit{TypeInv}$

$\mathit{GC} \triangleq$ INSTANCE *GeneralConsensus*

THEOREM $\mathit{Spec} \Rightarrow \mathit{GC}!\mathit{Spec}$