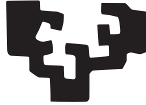


Universidad del País Vasco Euskal Herriko Unibertsitatea

University of the Basque Country UPV/EHU
Department of Computer Architecture and Technology

Distributed Eventual Leader Election in the Crash-Recovery and General Omission Failure Models

Dissertation
for the degree of Doctor of Philosophy

Candidate
Christian Fernández-Campusano

Supervisors
Roberto Cortiñas
Mikel Larrea

2019

(c)2020 CHRISTIAN FERNANDEZ CAMPUSANO

Abstract

Distributed applications are present in many aspects of everyday life. Banking, healthcare, or transportation are some examples of such applications. These are built on top of distributed systems. Roughly speaking, a distributed system is composed of a set of processes which collaborate among them to achieve a common goal. When building such systems, designers have to cope with several issues, such as different synchrony assumptions and failure occurrences. Distributed systems must ensure that the delivered service is trustworthy.

Agreement problems compose a fundamental class of problems in distributed systems. Agreement problems follow a similar pattern: *all processes must agree on some common decision*. Interestingly, most of the agreement problems can be considered as a particular instance of a problem called *consensus*, and, as a consequence, they can be solved by reduction to consensus. However, there exists a fundamental impossibility result, namely *(FLP)*, which states that in an asynchronous distributed system, it is impossible to achieve consensus deterministically when at least one process may fail. A way to circumvent this obstacle is by using *unreliable* failure detectors. A failure detector is an abstraction that allows encapsulating synchrony assumptions of the system and provides (possibly incorrect) information about process failures. A particular failure detector, called *Omega*, has been shown to be the weakest failure detector for solving consensus with a majority of correct processes. Informally, Omega lies in providing an eventual leader election mechanism.

In this work, we propose a distributed eventual leader election service prone to concurrent crash-recovery and omissions failures. Additionally, this work presents a performance study of consensus algorithms in omission and crash-recovery scenarios. The main contributions of this work are: *(i)* a novel definition of an eventual leader election service in distributed systems prone to failures of computation and connectivity, *(ii)* a specification of a weak system model for a distributed eventual leader election service, and *(iii)* three implementation approaches of a distributed eventual leader election service (*Basic, Communication-Efficient and Indirect-Leader-Trusting Mechanism*).

Acknowledgements

To my dear mother (RIP two years ago), a brave and courageous woman.
To my dear father, wise and consequent.
To my dear sisters and brothers, survivors of the dictatorship in Chile.
To my dear friends…
To my dear advisors…

Preface

This dissertation presents the results from the research carried out in the Distributed Systems Group at the University of the Basque Country UPV/EHU. Additionally, the part related to Distributed Eventual Leader Service was developed in collaboration with Michel Raynal, researcher member of the Institut Universitaire de France & IRISA.

The published articles related to this thesis are listed below:

- **A Performance Study of Consensus Algorithms** (Chapter 3).
  - Conference papers:
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea. *Improving the TrustedPals Framework Using Paxos*. PDP 2013: Work in Progress Volume, pp. 15-16 (CORE C).
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea: *Boosting Dependable Ubiquitous Computing: A Case Study*. UCAmI 2013: 42-45 (WoS).
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea: *A Performance Study of Consensus Algorithms in Omission and Crash-Recovery Scenarios*. PDP 2014: 240-243 (CORE C).
  - Journal papers:
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea: *Boosting Dependable Ubiquitous Computing: A Case Study* . IEEE Latin America Transactions: Vol. 12, No. 3, pp. 442-448 (2014) (JCR).

- **Distributed Eventual Leader Election Service** (Chapter 4).
  - Conference papers:
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea: *A Leader Election Service for Crash-Recovery and Omission Environments.* UCAmI 2014: 320-323 (WoS).
    - Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, Michel Raynal: *Eventual Leader Election Despite Crash-Recovery and Omission Failures.* PRDC 2015: 209-214 (CORE B).
    - Christian Fernández-Campusano, Roberto Cortiñas, Mikel Larrea, Jian Tang: *Designing and Evaluating Fault-tolerant Leader Election Algorithms.* PDP 2015, Work in Progress Volume, pp. 3-4 (CORE C).
    - Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, Michel Raynal: *A Communication-efficient Leader Election Algorithm in Partially Synchronous Systems prone to Crash-Recovery and Omission Failures.* ICDCN 2016: 8:1-8:4 (CORE B).
  - Journal papers:
    - Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, Michel Raynal: *A Distributed Leader Election Algorithm in Crash-Recovery and Omissive Systems.* Information Processing Letters 2017, Volume 118: 100-104 (JCR).

Contents

Abstract i

Acknowledgement iii

Preface v

Contents viii

List of Figures ix

List of Tables xi

List of Algorithms xiii

1 Introduction 1
1.1 Motivation 1
1.1.1 Aim of the Thesis 4
1.2 Contributions 5
1.3 Thesis Outline 6

2 Background 7
2.1 Distributed Systems 7
2.1.1 Concurrent and Distributed Computing 8
2.1.2 Safety and Liveness Properties 9
2.1.3 Types of Distributed Systems 9
2.2 System Models for Distributed Systems 11
2.2.1 Processes 11
2.2.2 Distributed Algorithms 12
2.2.3 Communication Links 13
2.2.4 Time and Timing Models 15
2.2.5 Process Failure Models 18
2.2.6 The Environment and Non-Determinism 21
2.3 Distributed Agreement Problems 22
2.3.1 The Non-Blocking Atomic Commitment Problem 22
2.3.2 The Consensus Problem 23
2.3.3 The Eventual Leader Election Problem 25
2.4 Unreliable Failure Detectors 26
2.4.1 Properties of Failure Detectors 26
2.4.2 Failure Detector Classes 28
2.4.3 The Omega Failure Detector 29
2.4.4 The Notion of Failure Detector Reduction 29
2.4.5 Approaches to Implementing Failure Detectors 31
2.5 Building Blocks for Fault-Tolerant Applications 31

3 A Performance Study of Consensus Algorithms **33**
3.1 The Study Context . . . 33
3.2 An Overview of the Considered Consensus Algorithms . . . 34
3.2.1 Chandra-Toueg's Algorithm . . . 34
3.2.2 Lamport's Paxos Algorithm . . . 35
3.2.3 Some Points of Comparison between both Algorithms . . . 37
3.2.4 Both Algorithms Solve Fault-Tolerant Agreement . . . 38
3.3 Case Study: Improving the TrustedPals Framework . . . 38
3.3.1 Solving Yao's Millionaire's Problem . . . 39
3.3.2 A Proposal for Improving TrustedPals . . . 40
3.3.3 System Model and Assumptions . . . 41
3.3.4 Architecture of TrustedPals . . . 42
3.3.5 Obtaining Ω_{Om} From a Simple Reduction of $\diamond\mathcal{P}_{om}$ . . . 42
3.3.6 A Novel Design of Architecture for TrustedPals . . . 44
3.4 A Practical Comparison of both Consensus Algorithms . . . 44
3.4.1 Related Studies . . . 45
3.4.2 Simulations . . . 45
3.5 Experimental Results . . . 47
3.5.1 Impact of Scalability on Average Latency . . . 47
3.5.2 Understanding the Behavior of Average Latency . . . 47
3.5.3 System under Multiple Simultaneous Failures . . . 51
3.6 Chapter Summary . . . 53

4 Distributed Eventual Leader Election Service **55**
4.1 Leader Election in Distributed Systems . . . 55
4.1.1 Synchrony and Failure Models . . . 56
4.1.2 Resilience to both Crash-Recovery and Omissions Failures . . . 56
4.1.3 About the Detectability of Failures . . . 57
4.1.4 Failure Detection and Leader Election . . . 57
4.1.5 Communication Efficiency . . . 58
4.2 System Model and Assumptions . . . 58
4.3 Specifying an Eventual Leader Election Service . . . 60
4.3.1 From Ω to an Eventual Leader Election Functionality . . . 61
4.4 Difficulties Underlying System Model Assumptions . . . 63
4.5 Implementing an Eventual Leader Service . . . 64
4.5.1 Basic Eventual Leader Election . . . 65
4.5.2 Communication-Efficient Eventual Leader Election . . . 71
4.5.3 Indirect-Leader Trusting Mechanism . . . 78
4.6 Chapter Summary . . . 86

5 Conclusions and Future Work **89**
5.1 Research Assessment . . . 89
5.2 Future Work . . . 91

References **93**

List of Figures

2.1 Steps of a process 12
2.2 The classical architecture of distributed systems 13
2.3 Representation of the main Timing Models 18
2.4 Failure Models in Fault-Tolerant Systems 20
2.5 Classification of Failure Detectors 28
2.6 Building blocks: Consensus and Failure Detectors 32

3.1 Current and Proposed Architecture for TrustedPals 43
3.2 The JBotSim Library 46
3.3 Impact of Scalability on Average Latency 48
3.4 Average Latency in Different Scenarios 50
3.5 Comparison for Different Numbers of Failures (*Crash scenario*) 51
3.6 Different Failures Scenarios 52

4.1 Example of three *connected* processes 64
4.2 Basic Eventual Leader Election *(Scenario of System)* 66
4.3 Communication-Efficient Eventual Leader Election (example) 73
4.4 Eventual Leader Election with an Indirect-Leader Trusting Mechanism (example) 79

List of Tables

2.1 Complexity vs. Computability 21

3.1 Improvement of Paxos+Ω_{Om} vs CT+$\diamond\mathcal{P}_{om}$ 49
3.2 Overhead of CT+$\diamond\mathcal{P}_{om}$ 49
3.3 Overhead of Paxos+Ω_{Om} 49
3.4 Complexity Improvement of Paxos+Ω_{Om} vs CT+$\diamond\mathcal{P}_{om}$ 51
3.5 Overhead of Paxos+Ω_{Om} vs CT+$\diamond\mathcal{P}_{om}$ (*multiples failures*) 52

Algorithms

4.1 Basic algorithm: process initialization. 66
4.2 Basic algorithm: main tasks. 67
4.3 Basic algorithm: *UpdateLeader()* procedure. 68
4.4 Communication-Efficient algorithm: process initialization. 73
4.5 Communication-Efficient algorithm: main tasks. 74
4.6 Communication-Efficient algorithm: *UpdateLeader()* procedure. 75
4.7 Indirect-Leader Trusting Mechanism: process initialization. 80
4.8 Indirect-Leader Trusting Mechanism: main tasks. 81
4.9 Indirect-Leader Trusting Mechanism: *UpdateLeader()* procedure. 82

1 Introduction

"We never stop reading, although every book comes to an end,
just as we never stop living, although death is certain."
— Roberto Bolaño[1]

In this chapter, we present the motivation to address the leader election problem in fault-tolerant asynchronous distributed systems. We aim at providing a distributed eventual leader election service prone to concurrent crash-recovery and omission failures. Usually, such services are used as a building block in fault-tolerant applications in distributed systems. We focus on devices that must operate despite holding limited resources of computation, communication, and storage. Finally, we present the contributions achieved in this work to provide such a service.

Outline. The rest of the chapter is organized as follows. Section 1.1 presents the motivation and aim of the thesis. The main contributions of this work are presented in Section 1.2. The chapter finishes in Section 1.3 with the structure of the thesis document.

1.1 Motivation

Nowadays, distributed systems are an essential piece in the information society. These systems are widely used in many aspects of everyday life. We live in a distributed world where distributed services arise at every moment. Distributed systems range from simple to highly specific applications, such as banking, healthcare, transportation, and air traffic control, among others.

Distributed systems execute their applications on a set of networked devices, and such distributed applications run the same pieces of code on different devices. Devices must be able to communicate in a collaborative way to achieve a common goal, regardless of their geographical location.

[1] Chilean novelist, short-story writer, poet, and essayist (1953-2003).

Although there is no single definition of a distributed system, we can mention two quotes formulated by well-known computer scientists to illustrate some inherent characteristics of such systems. On the one hand, paraphrasing to *George Coulouris*[2], he defined a distributed system as follows:

"A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing."

On the other hand, paraphrasing to *Leslie Lamport*[3], one of the foremost thinkers of the theory of distributed computing:

"A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable."

Informally, *we consider a distributed system composed of a set of processes that collaborate among them to achieve a common goal by exchanging messages.* Some authors consider other communication mechanisms, such as shared memory, but in this work, we focus on the message passing schema. In Chapter 2, we present the characteristics, definitions, and models of distributed systems.

Distributed computing covers a field of computer science that studies distributed systems. Distributed computing was born in the late seventies to take into account the intrinsic characteristics of physically distributed systems. Consequently, distributed computing arises to solve the problems concerning physically distributed entities (called process, node, sensor ...), in such a way that each entity only has some partial knowledge of the input parameters of the problem to be solved, and its local computation outputs may depend on some non-local input parameter. Therefore, computer entities inevitably have to exchange information and cooperate [Ray13]. In our case, we denote by *processes* the physically distributed entities in a distributed system. A process is considered an instance of an executed computer program, and it contains the program code and its current state. Most distributed systems allow many processes to execute concurrently. Simultaneous execution of such processes in a system may be altered by some of them stopping working, either by suffering a crash, a crash with a later recovering, or disconnections from the network. However, some of them can stay alive and continue to operate. These are the correct processes that remain operative and ensure minimal connectivity to achieve synchronization and consistency.

Dependability [Lap87, Lap92] is the ability to deliver a reliable service. The system suffers a failure when the service that is provided does not meet its specifications. Consequently, the dependability property is defined as the ability that has a distributed system to deliver trusted

[2]British computer scientist, `http://www.coulouris.net`
[3]American computer scientist, "http://www.lamport.org"

services. Dependability in distributed systems can be represented by several attributes, such as *availability, reliability, safety, security, maintainability, among others*. Both availability and reliability are critical attributes in distributed applications. Availability can be defined as the ability of a system to deliver a correct service, regardless of failures. Reliability can be defined as the ability to guarantee the continuous delivery of the correct service.

A distributed system is *reliable* if its behavior is predictable despite partial failures and asynchronous time periods. Achieving reliability requires robust, highly available, and reliable distributed applications, i.e., guaranteeing that the applications provide correct services despite failures on devices or processes or communication links. Thus, to provide high availability and reliability in a distributed system, it is required *fault prevention and fault tolerance*.

Fault-tolerant systems are usually based on the principle of redundancy. Redundancy implies the replication of critical components (hardware, software, or both) in a system to increase reliability. The replication involves sharing information to ensure consistency among the redundant components, as a way of improving reliability and fault tolerance in the system. The redundancy in a system is necessary to tolerate faults, but it is not enough, as shown in [Gär99]. As a result, detecting and correcting faults are also required for building dependable applications.

In distributed systems, consistency in the replication of critical components is ensured by using the agreement problem. *Distributed agreement problems* [PSL80, PT86, BJ87, BM93, Ray10] compose an essential class of problems in distributed systems. All agreement problems follow a similar pattern: *every correct process must agree on the same decision*. Reaching agreement in a distributed system is a fundamental issue of both theoretical and practical importance. Consensus, Atomic Commitment, Atomic Broadcast, Group Membership are different versions of the agreement problems in fault-tolerant dependable distributed systems [CB01, DSU04]. Several agreement problems can be solved by reducing them to a consensus problem.

The distributed consensus problem [PSL80, LSP82, SL84, DDS87, BM93] constitutes a paradigm that represents a family of agreement problems, and many agreement problems can be specified as a variant of the consensus problem. Informally, the consensus processes propose an initial value, and they have to agree on one of the proposed values. *The leader election problem* [MOZ05, GR06, Ray07, AJR10, LMA11, FLCR17] allows solving several agreement problems in distributed systems among them the consensus problem, e.g., the Paxos consensus algorithm [Lam98, PLL00, Lam01]. The Paxos algorithm is currently used in the computer industry (e.g., *Amazon, Google, IBM, Microsoft, VMware ..., among others*). In broad outlines, solving the leader election problem consists in providing each process in the system with a service such that: *(i) returns a process each time it is called*, and *(ii) always returns the same correct process*. A leader election mechanism allows boosting fault-tolerant distributed services since it provides support for building dependable distributed systems on specific timing assumptions.

The timing assumptions are fundamental to specify the study model since it allows making assumptions about the time and order of events in a distributed system [Lam78]. In the literature, there are two well-defined timing models for distributed systems [FLP85, HT93, Cri96, CGR11]: the *synchronous* and *asynchronous* models. The *synchronous model* assumes that there are upper

bounds on time, on the transmission delay of messages, and on the relative process speeds, and those bounds are known. In contrast, in the asynchronous model, there are no timing assumptions for the transmission delay of messages nor for the relative speeds of processes.

Fischer, Lynch and Paterson presented *an impossibility result* [FLP85], which states that it is impossible to solve agreement problems deterministically in an asynchronous model where at least one process may fail (known as *FLP impossibility result*). Nevertheless, several proposals have been presented to solve the distributed agreement problems on the synchronous model. Given this impossibility, some researchers focused their work on identifying the minimum amount of synchrony needed to solve agreement problems in the presence of failures.

Partially synchronous models were studied and presented by [DDS87, DLS88], such systems were refined over time, starting by [CT96]. Dwork et al. [DLS88] considered two types for a partial synchrony model. In the first, the timing attributes are bounded, but the bounds are unknown. In the second, the timing attributes are bounded and known, but they hold only after an unknown stabilization interval. Further, in [DLS88], it is shown that the consensus problem could be solved in both models, as long as there is a majority of correct processes in the system.

Later, the *unreliable failure detectors* were proposed by Chandra and Toueg [CT96] to circumvent the FLP impossibility result on the asynchronous model. Roughly speaking, an unreliable failure detector is an abstract module located at each process of the system, that allows encapsulating the synchrony assumptions of the system. A failure detector provides (*possibly incorrect*) information about (*the operational state*) of the other processes in the system. Chandra and Toueg in [CT96] defined eight classes of failure detectors, and they were classified according to the accuracy of the provided information. The failure detectors must guarantee two properties: *completeness and accuracy.* The consensus problem could be solved by using whatever the unreliable failure detectors introduced in [CT96].

Among the different classes of failure detectors that have been proposed, we consider particularly interesting the *Omega failure detector* (Ω) [CHT96], since it is the weakest failure detector that allows solving the distributed consensus problem, *assuming a majority of correct processes* in the system. The specification of the failure detector class Omega states that: *there is a time after which all the correct processes trust the same correct process.* Accordingly, the Omega failure detector allows providing an eventual leader election functionality [LMA11].

A distributed eventual leader election service provides a single process for coordination actions, allowing to keep a replicated system in a consistent manner despite failures. The scope of this thesis is to provide a service as support to solve distributed consensus problems in partially synchronous systems prone to failures of computation and connectivity.

1.1.1 Aim of the Thesis

We present the aim of this thesis by answering the following questions:

What do we want to do?

To provide a specification and an implementation
of an eventual leader election service
in partially synchronous distributed systems
prone to concurrent crash-recovery and omission failures

How do we begin our work?

We propose a performance study of consensus algorithms in omission and crash-recovery scenarios to provide us a better understanding of the leader election mechanisms in the consensus.

What are our perspectives at the end of the work?

Strictly speaking, from a theoretical to a practical perspective in partially synchronous distributed systems:

(i) Theoretical perspective: chasing a weaker system model to implement an eventual leader election service prone to both crash-recovery (*computation*) and message omission (*communication*) failures.

(ii) Practical perspective: proposing efficient algorithms for an eventual leader election service prone to both crash-recoveries and message omissions failures, based on a weak system model.

1.2 Contributions

In this work, we make some contributions to the state-of-the-art to provide a distributed eventual leader election service as support to solve consensus problems, in the following aspects:

- We conduct a performance study of distributed consensus algorithms in a partial synchrony model, under scenarios in which a process suffers both message omissions and crash-recovery failures. We use the TrustedPals framework [FFP⁺06, CFGA⁺12] as a case study. Initially, we analyze two well-known consensus algorithms: Chandra-Toueg (*round-based*) versus Paxos (*leader-based*). Also, we propose a new architecture for the TrustedPals framework, which is composed of a Paxos algorithm improved with the help of an eventual leader election service (Ω_{Om}). In order to support the proposal, we present a performance study (by simulation) under different scenarios of failure, mainly where processes are prone to suffer both omission and crash-recovery failures. In the practical comparative study, we use metrics to analyze their performance as the time complexity and the message complexity.

- We present a new definition of a distributed eventual leader election service, based on an Omega failure detector prone to failures of computation and communication. This new definition allows some "*incorrect*" processes to participate in the agreement of a common leader and take an active part in solving the consensus problem. For which:

  (i) We define a weak model to support the new definition of Omega, denoted $Omega_{crash-recovery,\ omission}$.

  (ii) We specify the properties of an eventual leadership that allows providing a distributed eventual leader election service.

- We provide three approaches that implement our proposal of a distributed eventual leader election service (*Basic, Communication-Efficient, and Indirect-Leader Trusting Mechanism*), in a partially synchronous model prone to concurrent crash-recovery and omissions failures. These approaches show a progressive weakening for the implementation of an eventual leader election service. Our algorithms tolerate the occurrence of crash-recoveries and message omissions to any process during some finite but unknown time, after which a majority of processes in the system remains up and does not omit messages.

1.3 Thesis Outline

This document is structured as follows:

Chapter 2. We present the background related to a distributed eventual leader election service working on a partially synchronous model prone to concurrent crash-recovery and omission failures.

Chapter 3. A performance study of Chandra-Toueg and Paxos consensus algorithms is presented. Also, a case of study is presented through a framework called TrustedPals. The previous architecture of TrustedPals uses the Chandra-Toueg consensus algorithm adapted to the general omission failure model. As an improvement, we propose using the Paxos consensus algorithm, in order to extend the applicability of this framework to the crash-recovery failure model as well. Additionally, we present a practical comparison by simulation of both architectures, highlighting the most significant results.

Chapter 4. We provide a specification for a distributed eventual leader election service in partially synchronous models prone to both crash-recovery (*computation*) and message omission (*communication*) failures. More precisely, in those systems, any process can fail as long as most processes meet some weak connectivity and reliability conditions. We also provide three different implementations of the specified distributed eventual leader service (*Basic, Communication-Efficient, and Indirect-Leader Trusting Mechanism*).

Chapter 5. We summarize the most significant results obtained from the carried research on specifying and implementing a distributed eventual leader election service, as well as suggested directions for future work.

2 Background

"Those who cannot remember the past are condemned to repeat it."
— George Santayana[2]

In this chapter, we present a background related to distributed eventual leader election. We expose the importance of agreement problems in asynchronous distributed systems prone to failures. Among them, we focus on the consensus problem and how it can be solved through an eventual leader election service. Such a service is essential for building fault-tolerant dependable applications in distributed systems.

Outline. This chapter is organized as follows. First, Section 2.1 describes distributed computing and its relationship with concurrent and distributed systems. In Section 2.2, we present some inherent characteristics of a distributed system. Distributed agreement problems and the most representative classes of those problems are described in Section 2.3. Section 2.4 presents the *unreliable failure detector* abstraction and how it is used to solve some distributed agreement problems, such as consensus. In Section 2.5, we conclude the chapter by describing how distributed services can be used as building blocks for dependable applications.

2.1 Distributed Systems

In nowadays, computing applications collaboratively execute pieces of code in devices of any kind, geographically distributed and interconnected by wired or wireless networks. A distributed system may be considered as a set of independent computing devices that are used jointly to make a particular task or to implement a distributed service [VR01, TS06], to reach a common goal. The properties of a distributed system characterize its behavior, and such properties are defined in a specific system model.

A classic distributed system is made up of n sequential deterministic processes, denoted p_1, p_2, $p_3 \ldots p_n$, with $n > 1$. These processes communicate and synchronize through a communication

[2]Spain-American philosopher, essayist, poet, and novelist (1863-1952).

medium, either a network that allows processes to send and receive messages, or a set of atomic read/write registers. Note that in this work, we only consider message-passing communication, i.e., by sending and receiving messages).

Processes in distributed systems collaborate to exhibit some helpful behavior according to the functionality of the distributed application. Consequently, they must meet the specifications of a distributed algorithm.

2.1.1 Concurrent and Distributed Computing

The study of concurrent computing [Dij65, Dij74, Lam74, AS83] started in the 1960s, with the presentation of the mutual exclusion problem by Dijkstra in [Dij65]. In computer science, concurrency indicates a decomposability property. Informally, the concurrent computing is a way of computation in which several calculations are executed over overlapping time periods (*concurrency*) rather than sequentially.

Observe that here, "*deterministic*" means that the behavior of a process is entirely determined from its initial state, the algorithm it executes, and (according to the communication medium) the sequence of values read from atomic registers or the sequence of received messages. Consequently, when different sequences of values are read, or messages are received in a different order, this can produce different behaviors.

A typical pattern of concurrent systems is that the processes interact with each other during the execution of some computation. Therefore, the number of possible forms of execution in the system can be considerably high, and the resulting outcome can be non-deterministic. Traditionally, concurrent and distributed systems are modeled through the use of non-deterministic transitions, i.e., for some state and input, the next state can be nothing or one or two or more possible states. Accordingly, a distributed system executes concurrent applications. The difference between concurrent and distributed systems is how the system behaves in the presence of failures, and this is widely studied in distributed systems.

Distributed computing was born to take into account the intrinsic characteristic of physically distributed systems [Lam78]. Accordingly, distributed computing looks for solving problems that involve physically distributed entities (called processes), such that each entity *(i) only has a partial knowledge of the many entries of the problem to be solved, and (ii) the computation of their local outputs could depend on some non-local inputs.* Consequently, computing entities have to exchange information and collaborate necessarily [Ray13].

A distributed system is affected by its timing pattern and the set of failures in which the system may evolve. Interestingly, a system does not master those characteristics but suffers it [Ray14].

2.1.2 Safety and Liveness Properties

Usually, to prove the correctness of concurrent algorithms, two classes of properties are used: *Safety and Liveness* [Lam77, AS85]. Both properties are often adopted in the design and specification of fault-tolerant distributed systems.

Identifying safety and liveness properties provides a better understanding of the problem, better design and specifications, and, consequently, clear and well-founded correctness proofs. Formally, safety and liveness guarantee the following properties:

(i) The safety property states that some particular *bad thing never happens*. This property ensures that the algorithm should not do anything wrong, i.e., a safety property *is continuously met*.

(ii) The liveness property states that some particular *good things will eventually happen*, i.e., the algorithm will eventually produce a result. In other words, a liveness property *will eventually be met*.

Differentiating between safety and liveness properties is crucial to prove that a program fulfills a set of logical rules for reasoning rigorously about its correctness. Informally, a safety property involves an *invariance argument*, while a liveness property involves a *well-founded argument*.

More information concerning safety and liveness properties can be found in [Lam77, Lam79, AAH+85, AS85, AS87, CBTB00, BFH+06]

2.1.3 Types of Distributed Systems

Distributed systems can be categorized into the following classes:

2.1.3.1 Classical Distributed Systems

In general, a classical distributed system [Lam78, CGR11, Ray13] is composed of a fixed and known quantity of processes (denoted n). Two processes do not have the same identity, and each process knows the identity of all the other processes.

Usually, the communication network is fully connected. Further, depending on timing assumptions, there is, or there is not an upper bound on time of message delays and the speed of processes.

When considering classical distributed systems prone to failures, another fundamental parameter is the maximal number of faulty processes (denoted f). In classical distributed systems, n and f are known. From the point of view of protocol design, a process can safely use these parameters. Classical distributed systems are also known as static distributed systems.

2.1.3.2 Dynamic Distributed Systems

Dynamic distributed systems [MRT+05, BBRP07, LRAC12], in general terms, share a common characteristic: *processes can join and leave the system at will.*

Nevertheless, there is a great need for the definition of such models to develop trusted services suited to dynamic environments. Informally, a dynamic distributed system is continually executed by a large number of processes, but any process can leave (or return) to interact with a part of the whole system, at any time. Unlike static distributed systems, managing dynamic systems requires to capture the notion of eventual stability.

2.1.3.3 Homonymous Distributed Systems

In homonymous distributed systems [DGFG+11, AAI+15, JAT15], the homonymy is a generalization of two cases: *(i) the system has unique identifiers for every process (unicity), and (ii) the system has the same identifier for all processes (anonymity).* These cases are the two extremes of homonymy.

Consequently, distributed systems can also be classified by the identity of their processes, as described below:

- *Unique non-anonymous distributed system.* Each process has its unique identification number, and its features are also unique.
- *No-unique non-anonymous distributed system.* Several processes may have the same identifier, i.e., the system has a limited number of authenticated identities, and at least two processes share the same identification number.
- *Anonymous distributed system.* All processes have the same identification number and the same features. The processes are anonymous, i.e., there is no way to differentiate between two processes in the system.

Observe that both systems, *(i) unique non-anonymous identifiers and (ii) anonymous identifiers*, are extreme cases of homonymous systems.

Anonymous processes are usually practical in some distributed systems, such as sensor networks, where assigning a unique identifier for each device is not always possible. Another practical scenario is related to privacy problems (e.g., to hide the identity of a user in a system).

However, most of the algorithms proposed for classical distributed systems with unique identifiers do not work correctly in the presence of identification collisions. It has been widely studied that systems with different process identifiers can solve more problems or can perform better than anonymous systems (where all identifiers are implicitly the same).

As a conclusion, the leader election problem cannot be solved in anonymous systems.

2.2 System Models for Distributed Systems

Roughly speaking, a system model is a collection of assumptions that allow defining characteristics and constraints that a system might exhibit, such as processes classification, communication links, timing model, or failure patterns. To design an algorithm, one of the critical tasks is defining the system model in which the algorithm must work. Consequently, to implement an algorithm in a real system, it will be sufficient to know if that real system satisfies the properties of the system model for which the algorithm was designed.

Models are a fundamental tool for designing and implementing solutions to a given problem since, by simplification, formulate the object of study and propose a set of rules to define its behavior. Consequently, a model is an abstraction of an object of interest. In [Sch93], it was established that a good model must be *accurate* and *tractable.* A model is *accurate* if its analysis offers a high degree of trust in the object of interest. On the other hand, a model is *tractable* if its analysis is possible.

A distributed system can be modeled by describing its desirable behavior. First, it is necessary to present the following concepts:

- State. The state of a distributed system is constituted by a state set of its components (i.e., processes and communication channels).
- Trace. The behavior of a distributed system is represented by a sequence of states (all possible executions). A trace is a possible execution of the system.
- Step. A pair of successive states, i.e., possible transitions between states.

Given that the behavior of a system is represented through traces, a *property* can be defined as a collection of such traces, and it represents a component or the distributed system as a whole. Defining the properties of the system is a useful technique since it allows capturing abstractions regardless of their implementation – for example, *safety* and *liveness* properties.

Below, we present the different components we will use to model a distributed system, such as processes, communication links (also communication channels), time and time models, failure patterns, and the relationship between the environment and non-determinism.

2.2.1 Processes

A process can be defined as an abstract computational entity able to execute computations in a distributed system. In this way, every process executes a copy of the same distributed algorithm, keeping a local state of that execution without loss of generality. We consider distributed systems composed of a finite set of processes (Π), where $\Pi = \{ p_1, p_2, p_3 \ldots p_n \}$ with $n > 1$. Processes interact by exchanging messages trough communication links.

In a distributed system, a process can be considered correct or incorrect depending on its pattern of failure (see Section 2.2.5):

- ***Correct process.*** It is a process that behaves according to its specification, i.e., a process that does not suffer any failure. We denote the number of correct processes in the system by c.
- ***Incorrect process.*** It is a process that suffers a failure that might affect its specifications, e.g., a non-correct process. We denote f the number of non-correct processes.

Observe that the number of correct processes is $c = n - f$ in a system with n processes.

2.2.2 Distributed Algorithms

A distributed algorithm is expressed through a collection of deterministic automata [Ray13], one per process. The execution of such an algorithm is represented by a sequence of steps that the system processes execute. In general terms, a run of an algorithm is the combination of the followed steps and state information of a given execution.

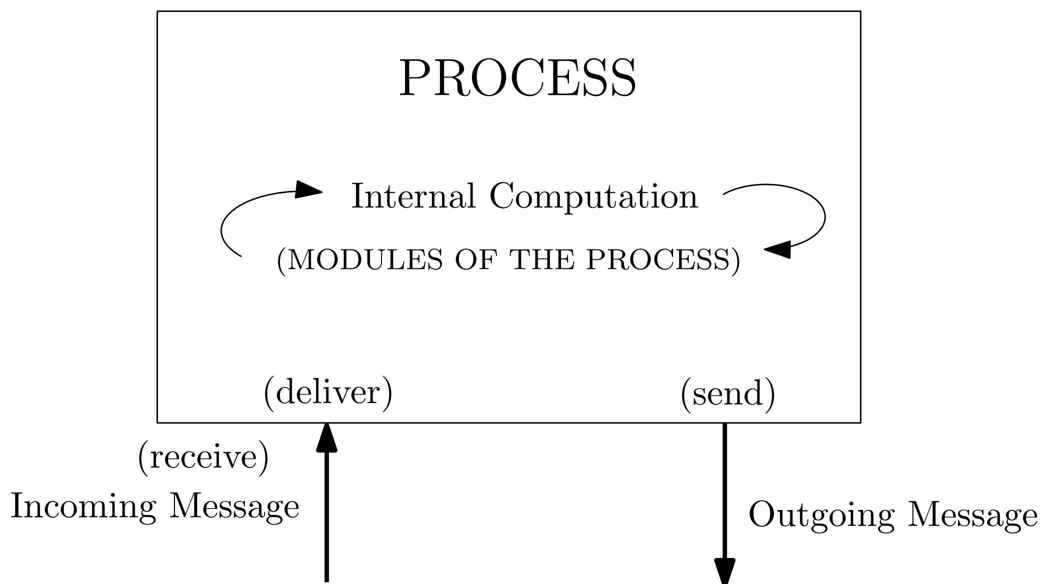


Figure 2.1: Steps of a process

Figure 2.1 presents a sketch of *process steps*, wherein a process delivers a message from another process, executes a local computation, or sends a message to some other process. In a more detailed way, the figure presents an execution step for a process, and it can be explained as follows. A process takes an event from a particular queue, it executes a state transition according to that event, and later it may send a message, or it appends a new event to the appropriate queue. Similarly, when a message arrives, it is treated as an event, i.e., when a message arrives, an event is added to the queue. A message is received by the process when such an event is processed. Consequently, a process which does not fail can execute infinitely many events.

The event generation in distributed algorithms can be classified into two classical approaches [ZWD+14]: *time-driven* and *message-driven*. In *time-driven* algorithms, the events can be triggered using the passage of time (measured by clocks). Instead, in *message-driven* algorithms, the generation of events is only based on the reception of messages.

2.2.3 Communication Links

Communication links or channels represent a high-level abstraction for a distributed network. The network allows communication between heterogeneous computing devices (processes) that compose the distributed system. Consequently, processes communicate through the exchange of messages.

The architecture of a classical distributed system is presented in Figure 2.2, wherein processes communicate through a network by communication links. As shown in Figure 2.2, we differentiate two atomic operations for message delivery: *receive* and *deliver*.

First, receiving data (*receive* operation) implies that a message arrives at the target process. Then, delivering data (*deliver* operation) is the following step, which precedes the third step, processing the message. Observe that the message could be omitted after the receive operation and before the delivery operation. Note that each sent message is marked with a unique identifier.

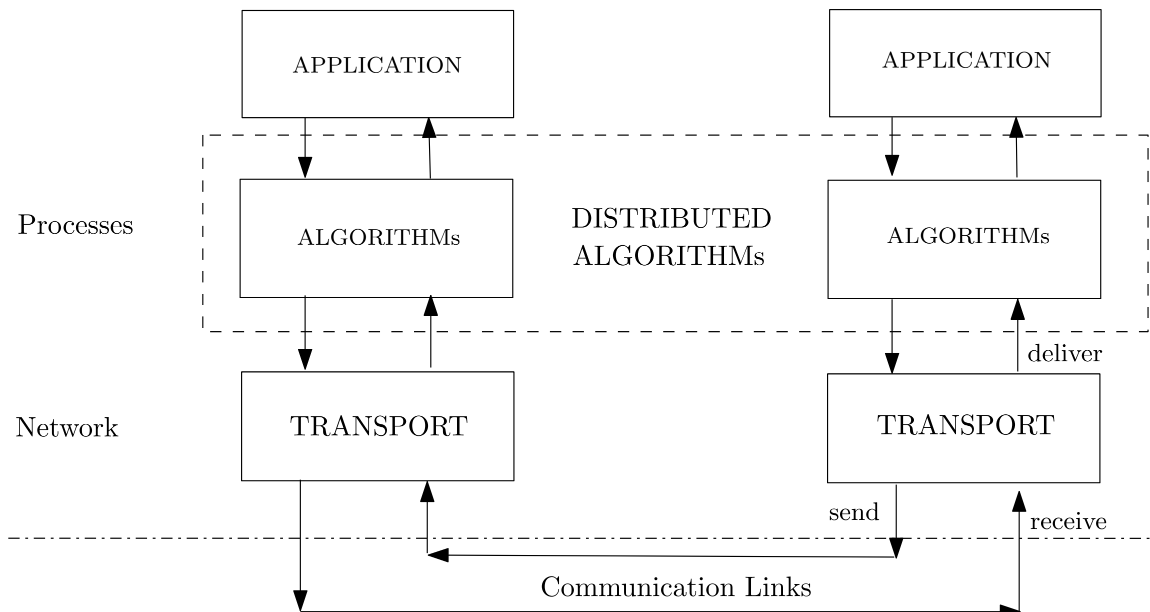


Figure 2.2: The classical architecture of distributed systems

Regarding the directionality of communication links, we consider two types: *unidirectional* and *bidirectional.* A unidirectional communication link from a process p to another process q allows p to send messages to q (only one-way). When a link from p to q is bidirectional, it is assumed that the communication link allows sending messages from p to q and from q to p (two-way).

Communication links can meet different properties. Therefore, they can be classified into well-defined types [BCT96, ADFT08, CGR11]. On the one hand, On the one hand, the so-called *Reliable* link can be built over mechanisms to detect and suppress message duplicates, plus mechanisms for message retransmission. The main problem is that reliable communication links cannot always be provided. On the other hand, Basu, Charron-Bost and Toueg [BCT96] consider two models of lossy links: *Fair-Lossy* and *Eventually Reliable.*

2.2.3.1 Reliable Links

Reliable communication links [BCT96, CGR11, Ray13], also denoted as *perfect links*, ensure that each message sent to a correct process is received, i.e., there is no message loss. Further, no message is created, corrupted, or duplicated by the communication link.

Formally, *reliable links* guarantee the following properties:

- *No creation.* If a message m is received by a process q, then m was previously sent to q by another process p.
- *No duplication.* No message is received more than once.
- *No loss.* If a process p sends a message m to another process q and q is correct, then q will eventually receive m.

The “no creation" and “no duplication" properties are *safety* properties, whereas “no loss" is a *liveness* property.

2.2.3.2 Fair-Lossy Links

A fair-lossy communication link [BCT96, CGR11, Ray13] ensure that, if an infinite amount of messages is sent, then it may drop (loss) an infinite number of messages, but an infinite subset of messages will be received. As a consequence, if a message is sent infinitely often, then the receiver will eventually receive it.

Formally, a *fair-lossy link* must satisfy the following properties:

- *No creation.* If a message m is received by a process q, then m was previously sent to q by another process p.
- *Finite duplication.* If a message m is sent a finite number of times by a process p to a process q, then m cannot be delivered an infinite number of times by q.
- *Fair loss.* If a process p sends a message m to a correct process q an infinite number of times, then q eventually receives m from p.

Observe that a problem in the crash failure model with *reliable links* can be simulated using *fair-lossy links* and a majority of correct processes [BCT96]. Therefore, any problem solvable by using *reliable links* is also solvable by using *fair-lossy links*.

2.2.3.3 Eventually Reliable Links

An eventually reliable communication link [BCT96, CGR11, Ray13] can drop (loss) messages (like a *fair lossy link*), but there is a time after which all sent messages are eventually received (like a *reliable link*). Observe that an eventually reliable link can lose an unbounded but finite number of messages.

Formally, an *eventually reliable link* must satisfy the following properties:

- *No creation.* If a message m is received by a process q, then m was previously sent to q by another process p.
- *No duplication.* No message is received more than once.
- *Finite loss.* If q is correct, the number of messages sent by p to q that are not received by q is finite.

Reliable links are strictly stronger than *eventually reliable links* [BCT96], i.e., some problems solvable with *reliable links* may not be solved with *eventually reliable links.*

2.2.4 Time and Timing Models

In this section, we examine the time assumptions and its critical impact on distributed systems. The time in distributed systems reflects many of the characteristics which underlie the system [Lyn96], as the computation time on a process (computation process time) and time communication delays between processes (message delay time). There are various ways to model uncertainties in the system and to cope with them.

First, we will introduce the concept of the clock as an abstraction for measuring the passage of time in a system, and later we will present the different models of time in distributed systems, i.e., *synchronous*, *asynchronous* and *partially synchronous* models.

Clocks. In a distributed system, each process has a *logical clock* that enables measuring time passage, although it is not always possible that every clock remains synchronized with the rest of the clocks. Lamport [Lam78] proposed the concept of *logical time*, which allows the time management in an abstract way. An event (also denoted as a *step*) can be considered as an atomic execution (action or instruction) of an algorithm. Consequently, by using the *logical time* it is possible to augment every process with a *logical clock* and thus to enable to order events in distributed systems.

Global Clock. The representation of the global time in the model can be simplified by using a discrete global clock. However, a process does not have access to this clock (it is a fictitious device). Each event in a process is always associated with a specific global time, and usually, it is assumed as a linear model of event execution. In this regard, the time domain (denoted T) will always be represented by consecutive natural numbers.

2.2.4.1 Synchronous Model

The synchronous model [HT93] presents explicit bounds on time, i.e., a message is delivered in a bounded time, and the relative speed of computation in a process is known. Both are bounded and known.

Synchronous distributed systems are based on strong boundaries on message delays and computation time. The problem with this model is that it does not accurately reflect reality. The positive thing about this model is that, when used, it is possible to achieve theoretical results that can later be reduced to weaker models.

A synchronous system meets the following characteristics:

- The time to execute each step in a process is known and limited. There are upper and lower time bounds.
- Every sent message is received within a known limited time.
- Every clock has a known and bounded deviation from real-time.

The differentiation between a crashed process and a slow processes is possible due to deterministically detecting those failures in a synchronous system. Therefore, agreement problems (the consensus problem, among others) can be solved efficiently in synchronous systems.

Observe that the main problem of synchronous systems is the fact that the bounds have to be defined to behave correctly in the worst case. Otherwise, the assumptions of the system model might be violated, and therefore, the algorithm may lose its correctness.

2.2.4.2 Asynchronous Model

In the asynchronous model [FLP85, Cri96], there are no upper bounds on the transmission delay of messages, and on the relative speed of processes, i.e., there are no timing assumptions.

A difficulty that implicitly comes with this model is how to assure both safety and liveness properties in the system. The impossibility of solving the problem of consensus in a deterministic way when a process suffers a failure, known as the FLP impossibility result [FLP85], which relies on the fact that sometimes it is not possible to distinguish between a failed process or a process that is taking an infinitely long time to reply to a message.

Some of the advantages of this model are:

- It has a better adaptation to the rapidly growing and variable load on the networks, regarding processes and communication links.
- Algorithms that work in this model always preserve correctness (safety property), when they are executed in any other system model.

Several problems have been proved to be impossible to solve in asynchronous systems, given that when a failure occurs, there are problems to detect it in a deterministic way. For instance, let us suppose that a process p is waiting for a message m from another process q to know whether q has crashed or not. So, during its wait p does not have any way to know how long it should be waiting for the message before considering that there has been a failure (either a crash of q or omission of m). Then, there are two possible ways of making a mistake:

1. *The safety property of the application is violated.* It happens when p finally decides to consider q as faulty since the expected message has not been delivered yet. Suddenly, it might happen that the expected message finally arrives. Therefore, p made the wrong decision.

2. *The liveness property of the application would not be satisfied.* It happens when p decides to wait until the message is received. Then, it might happen that the message never arrives at p, e.g., q failed before sending it. Therefore, p made the wrong decision.

2.2.4.3 Partially Synchronous Models

In the partially synchronous model, processes have some knowledge about timing assumptions, as they may have access to almost-synchronized clocks, or they might have estimates of how long messages take to be delivered or how long it takes processes to execute a step. In real life, there exist systems that almost always behave like synchronous systems but can be unstable during some periods of time, behaving like an asynchronous system and thus exceeding normal bounds.

The partially synchronous model was presented in [DDS87] with the aim of defining an intermediate model between synchronous and asynchronous systems. In [DDS87], five parameters have been proposed to classify different partially synchronous systems, with 32 (2^5) different models. Regarding synchrony, four of those models were identified as minimal models in which consensus is solvable in partial synchrony.

In [DLS88], two models were considered :

- **Model 1.** There exist time bounds on message delays and relative speed of processes, but they are not known.
- **Model 2.** Bounds exist, and they are known, but they hold only after an unknown time called *Global Stabilization Time* (*GST*). This model is also-called *eventually synchronous.* Before *GST*, the system behaves like an asynchronous system, and after *GST*, it behaves like a synchronous system.

Later, another partially synchronous model was introduced in [CT96] with weaker assumptions than the previous ones (Model 1 and Model 2):

- **Model 3.** Bounds exist, but they are not known, and they only hold eventually.

Specifically, Model 3 is the partially synchronous model considered in our work. Moreover, an eventually synchronous system is not necessary to hold its bounds continuously but only requires that the system keeps stability during a sufficiently long time to complete the algorithm execution.

Another important aspect is that there is no single definition for partially synchronous models. The model of partial synchrony has found many refinements, beginning by Chandra and Toueg [CT96], where both communication and processes are partially synchronous (regardless of the model of partial synchrony, presented by Dwork et al. [DLS88]), followed by Hermant and Widder [HW05], where only the rate between best-case and worst-case round-trip delay is bounded. Also, Fetzer et al. [FSS05] worked with system models in which only an average response time is bounded.

2.2.4.4 Representation of Timing Models

The representation of the timing models is shown in Figure 2.3.

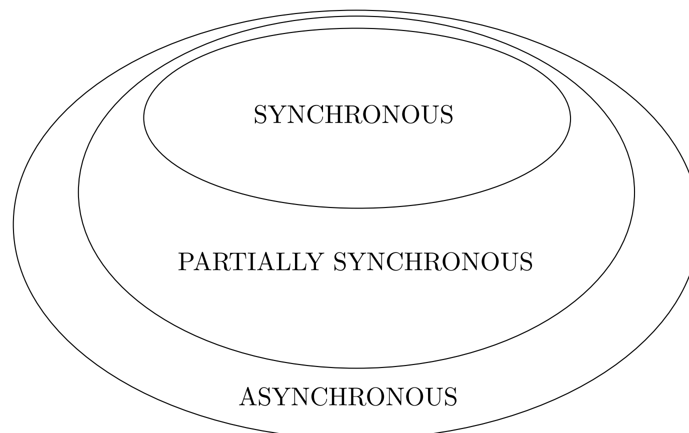


Figure 2.3: Representation of the main Timing Models

Hence, synchronous systems $\subset$ partially synchronous systems $\subset$ asynchronous systems.

2.2.5 Process Failure Models

An essential aim in the design of distributed applications is to build them on systems in a way that they can recover from some failures without harmfully affecting the overall performance, i.e., if a part of the system suffers a failure, then the remaining part continues to operate.

Bellow, we present a classification of failure patterns that can suffer a process in a distributed system. We will consider the following types of failures: *Crash-Stop*, *General Omission*, *Crash-Recovery*, and *Byzantine*.

2.2.5.1 Crash-Stop Failure Model

In the crash-stop model [CGR11, Ray13], also the crash model, the process that suffers a crash failure stops executing its algorithm, and thus it does not send and/or receive messages never again. Basically, in this model, a faulty process operates as a correct one before it crashes, and, after crashing, it remains inactive forever.

Observe that in synchronous systems, fail-stop processes can be efficiently implemented by using time-outs. A stronger type of crash, fail-stop, was proposed in [SS83]; when a process crashes, all correct processes are informed about that failure.

The crash-stop failure in the context of agreement problems (such as the consensus problem) was first proposed in [LSP82] and later extended in [HT93].

2.2.5.2 General Omission Failure Model

The omission failure model [CGR11, Ray13] occurs in the process and not in the communication links. This failure can describe situations such as buffer overflows or malicious dropping of messages. Recall that a data buffer is a sector in physical memory storage used to store data momentarily while it is moved to another side. For example, a buffer between a process and their communication link.

An omission failure can be classified into *send-omission* and *receive-omission*, as follows:

- A process p suffers a *send-omission* failure if it executes a send-message instruction, but the message never reaches the link.
- A process p suffers a *receive-omission* when a message is received at its destination process, but inside this process, the message is never delivered.

The *send-omission* failure model was proposed in [HT93] to define a failure model in which processes can crash (but cannot recover) and suffer *send-omission*. Perry et al. [PT86] proposed the general omission failure model by adding to the previous one the possibility of suffering *receive-omission* failures as well.

Further, we can distinguish between *permanent* and *transient omission* as follows:

- A process that suffers a *permanent omission* failure, after omitting a message, every subsequent message will be omitted.
- A process that suffers a *transient omission* failure, will reliably send or receive messages until another failure.

2.2.5.3 Crash-Recovery Failure Model

In the crash-recovery failure model [CGR11, Ray13], a crashed process can recover after some time. When a crashed process recovers, it can affect its ability to remember the previous it had before crashing, i.e., it may lose all pre-crash information, so it should have to start from scratch after recovering.

Sometimes, processes are augmented with stable storage, a special type of memoria that keeps stored information even if a crash occurs. Therefore, a crashed and recovered process could subsequently obtain essential information to recover its operational state correctly. However, stable storage is an expensive resource, so efforts have been made to use it as less as possible [HMR98, ACT00, MLJ09, CGR11].

2.2.5.4 Byzantine Failure Model

The Byzantine failure model, as defined in [LSP82], considers arbitrary failures, in which a process can make wrong state transitions and send arbitrary messages. Therefore, it affects the safety property since it is deviates from its algorithm specification.

This type of failure is also-called malicious, and it can be modeled as if an evil entity produced them. It is also contemplated that malevolent entities can collaborate through different processes.

For more information on the Byzantine Failure Model, see [LSP82, DGFG+11, Ray13].

2.2.5.5 Connecting Failure Models

Figure 2.4 represents a classification of the previously presented failure models (adapted from [BM93, CGR11, Ray13]).

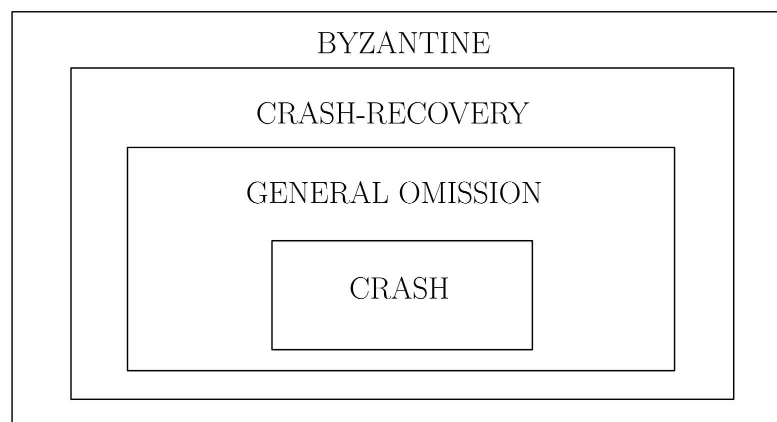


Figure 2.4: Failure Models in Fault-Tolerant Systems

The classification is possible because the containing set can simulate the failures of their subset:

- In the General Omission model, a crashed process experiences a similar external behavior like a process that suffers permanent send and receive omission.
- In the Crash-Recovery model, some omission failures can be simulated through a crash and recovery by using *stable storage*.
- In the Byzantine model, arbitrary failures include any other type of failure.

Under certain conditions and assumptions, each failure model may be considered as a subset of some failure model superior that includes it (i.e., *Crash* ⊂ *General Omission* ⊂ *Crash-Recovery* ⊂ *Byzantine*).

2.2.6 The Environment and Non-Determinism

The environment of a distributed system is composed of failures (processes and communication links) and the assumption of synchrony (from a synchronous model to an asynchronous model) in which the system may evolve. Therefore, a system does not dominate its environment but suffers it. The processes in distributed systems are deterministic, then the only non-determinism a distributed system has to cope with is the non-determinism produced by its environment.

A fundamental question in a distributed system is what can be computed. The answer to that interrogation depends on the environment in which the considered distributed system evolves, i.e., on the assumptions that the system is based on. Such an environment is very often left implicit and nearly always not formulated regarding their specific requirements. In the case where the environment is such that there is no synchrony assumption and the processes may suffer failures, many problems become impossible to solve (e.g., the consensus problem in asynchronous systems). Hence, it is essential to know the weakest assumptions (lower bounds) that set the limits beyond which the considered distributed problem cannot be solved.

Computability and complexity are the two perspectives that allow us to understand and study the distributed problems. Table 2.1 presents the main issues in distributed systems when contemplating these two perspectives [Ray14].

Table 2.1: Complexity vs. Computability

Environment	Failure-free	Failure-prone
Synchronous Model	Complexity	Complexity
Asynchronous Model	Complexity	Computability

Remark. Deterministic means that the behavior of a process is entirely determined from its initial state, the algorithm it executes, and the sequence of messages received.

2.3 Distributed Agreement Problems

In distributed systems, reaching agreement is a fundamental issue, both from theoretical and practical perspective. Agreement problems follow a common pattern: *all participating processes must agree on some common decision.* Examples of agreement problems are *Consensus* [PSL80, LSP82], *Atomic Commit* [Ray97, Gue02], *Group Membership* [BJ87, CKV01], *Totally Ordered Broadcast* [CT96, DSU04], and *Leader Election* [Ray13, FLCR17].

Essentially, the distributed agreement problem is a general problem where processes of a distributed system must agree on an abstract fact. Solving the distributed agreement problem is essential to implementing many of the current distributed applications. In this sense, the *distributed consensus problem* [PSL80, LSP82] depicts a family of agreement problems, since it represents the difficulty to solve such agreement problems in fault-tolerant distributed systems [SL84].

Unfortunately, solving the distributed agreement is complex; e.g., it cannot be solved in asynchronous systems prone to process failures. This result, known as the FLP impossibility result, was presented by Fischer, Lynch and Patterson [FLP85]. In the following sections, some solutions are presented to circumvent the FLP impossibility result.

The leader election problem is the focus of our study since it allows solving the distributed consensus problem. The election of a distributed leader involves determining a single correct process that coordinates the actions of all processes that make up the distributed system.

2.3.1 The Non-Blocking Atomic Commitment Problem

A fundamental problem in distributed systems is ensuring that data always remains consistent [Lam80]. The non-blocking atomic commitment problem [Gue95, Ray97, Gue02] consists of guaranteeing that all correct processes take the same decision, especially the commit or abort of a transaction in an atomic operation.

A transaction usually involves several sites (processes). At the end of a transaction, processes must participate in an agreement protocol to commit it (*everything went well*) or abort it (*something went wrong*).

A commitment protocol (an agreement protocol) called Two-Phase Commit obeys a two-phase pattern. First, each process votes either *yes* or *no*. If, for any reason (deadlock, storage problem, concurrency control . . .), a process cannot locally commit the transaction, it votes *no*. Otherwise, a *yes* vote implies that the process commits. The second phase declares the order to commit the transaction if all processes voted *yes* or to abort it if some process voted *no*.

2.3.1.1 Non-Blocking Atomic Commitment Properties

Basically, in the non-blocking atomic commitment problem, if the decision is committing, then all processes make their updates permanent. The commit/abort value of the outcome of the protocol depends on the votes of processes and failures. More precisely, this problem is defined by the following properties [Ray97]:

- *Termination.* Every correct process eventually decides.
- *Validity.* If a process decides to commit, then all processes have voted *yes.*
- *Integrity.* A process decides at most once.
- *Uniform-Agreement.* No two processes decide differently.
- *Non-Triviality property. S-Non-Triviality* or *AS-Non-Triviality.*

The non-triviality condition aims to eliminate "non-expected" states in which the decision does not depend on the failure scenarios and votes.

As failures can be reliably detected in synchronous systems, the Non-Triviality property for these systems is [Had90]:

- *S-Non-Triviality.* If all processes vote *yes* and there is no failure, then the outcome decision is to commit.

On asynchronous systems, failures can only be suspected, possibly erroneously, and the condition has to be weaker for the problem to be solvable [Gue95]. So, the Non-Triviality property for these systems is:

- *AS-Non-Triviality.* If all processes vote *yes* and there is no failure suspicion, then the outcome decision is to commit.

2.3.2 The Consensus Problem

The consensus problem [PSL80, LSP82, FLP85, DDS87, DLS88, CT96] is one of the most important problems in fault-tolerant distributed systems, and it constitutes a paradigm that represents a class of agreement problems. Accordingly, the consensus problem has been used to solve several agreement problems.

Informally, in consensus, processes *propose* an initial value, and, despite failures, correct processes have to *decide* the same unique value from the proposed values. Formally, the consensus problem is defined in terms of two primitives:

- *Propose(v).* Every process proposes an initial value (v).
- *Decide(v).* All (correct) processes decide on the same value (v), which must be one out of the initially proposed values.

In a synchronous system, a process can reliably detect if another process has failed by the use of time-outs. If the timer expires, it means that the other process has failed (assuming that communication is reliable). Hence, reliable detection of failures is possible, and the consensus problem can be solved.

Observe that consensus has a trivial solution in systems where processes do not fail. However, the agreement on systems where processes can fail can be much more difficult, since the resolvability of the consensus depends on a large extent on the possibility of detecting failures in the processes and their communication links.

Although many solutions have been proposed to solve consensus in *synchronous systems*, Fischer, Lynch and Paterson presented the *FLP impossibility* [FLP85], which states that it is impossible to solve consensus deterministically in *asynchronous systems* where at least one process may crash. This result generated a series of works that tried to identify the amount of synchrony needed to solve consensus in the presence of failures, e.g., in *partially synchronous systems* [DDS87, DLS88, CT96].

2.3.2.1 Consensus Properties

The consensus problem is defined by the following properties [CT96]:

- *Termination.* Every correct process eventually decides some value.
- *Validity.* If a process decides v, then v was previously proposed by some process.
- *Integrity.* A process decides at most once.
- *Agreement.* No two correct processes decide differently.

According to the definition of the *agreement* property, a faulty process may decide a different value from the one that correct processes agree on. Charron-Bost and Schiper show in [CS04] that *the uniform consensus problem* is harder than the consensus problem. The *uniform consensus* problem states that all (correct and faulty) processes that decide must decide the same value. The *uniform consensus problem* redefines the *agreement* property as *uniform agreement* property [CS04]:

- *Uniform Agreement.* No two processes decide differently.

Remark. Any algorithm in a partial synchrony model that solves the consensus problem also solves the uniform consensus problem [Gue95, GL08].

According to *safety* and *liveness* properties (see Chapter 2.1.2):

- *Uniform Agreement, Validity, and Integrity* are safety properties, whereas
- *Termination* is a liveness property.

Remark. Recall that a *safety* property establishes that the system will behave according to its specification (*ensures that something bad never happens*), and a *liveness* property states that the system will eventually make progress (*ensures that something good eventually happens*).

2.3.3 The Eventual Leader Election Problem

A correct process can be considered as a leader process if such a process assumes a coordinating role for every correct process in the system. Also, the leader process may suffer some failure at any time, in which case, it is necessary to apply a procedure for the eventual election of a new one [SM95].

In general terms, in the eventual leader election problem correct processes must eventually decide the same correct process as the single leader. As a result, the system processes can be classified into two sets: *leader* and *non-leader*.

The eventual leader election problem can be solved through a distributed leader election service with characteristics of eventuality in the time. A *distributed eventual leader service* can be defined as follows [Ray07]:

(i) Each time it is called, it returns the identity of a process (p_{id}), and

(ii) after some finite time, it always returns the same (p_{id}) corresponding to the identity of a *correct process.*

The interest in the study of this problem lies partly in the difficulty of solving it in the asynchronous system.

2.3.3.1 Eventual Leader Election Properties

More precisely, the eventual leader election problem is defined by the following properties [Ray07, Ray13]:

- *Termination.* Every correct process eventually decides some leader.
- *Validity.* If a process decides a leader, then this leader was previously proposed by some process.
- *Agreement.* Every correct process elects the same leader.

2.4 Unreliable Failure Detectors

An unreliable failure detector is an abstract module located at each process of the system that provides (*possibly incorrect*) information about (*the operational state of*) other processes in the system. The principal objective of this abstract module is to provide an abstraction of the timing assumptions of the system.

So, *how does a failure detector work?* Roughly speaking, the distributed systems designed with failure detectors provide each process with access to a local failure detector module, which monitors other processes in the system and maintains a set of suspects to have failed. A failure detector module can make mistakes by not suspecting a process that has failed or by wrong suspicions, i.e., it can suspect that a process p has crashed even though p is still running. If it later determines that suspecting p was a mistake, then it eliminates p from its set of suspects. Therefore, each module may attach and eliminate processes regularly from its set of suspected processes. Furthermore, the failure detector modules in two different processes may have different sets of suspected processes at any time.

Based on the literature, we can mention two types of failure detectors: *Perfect* and *Unreliable*.

(i) In a fully synchronous distributed system *Perfect failure detector*, i.e., a failure detector that does not make mistakes can be implemented in fully synchronous systems. In such systems, a simple timeout-based algorithm can reliably detect the failure of any process.

(ii) In a non-synchronous distributed system, a failure detector may not correctly infer about which processes have failed, and its type is defined as an *Unreliable failure detector*. Not surprisingly, the given information by the first type *(i)* about failed processes will be more accurate than the information provided by those of the category of unreliable failure detectors.

Unreliable failure detectors [CT96] can be viewed as an abstract way of incorporating partial synchrony assumptions into the model of the distributed system. Systems using these unreliable failure detectors are not purely asynchronous. It merely produces the vision of an asynchronous system through the encapsulating of references to time in the failure detector, in such a way that it allows solving the agreement problems in distributed asynchronous systems.

Clearly, in a synchronous system, the timing assumptions are stronger, allowing to provide more accurate information about process failures. For more information, see [CT96, MMR02, Ray05, Gue08, Ray10, FGK11, CGR11].

2.4.1 Properties of Failure Detectors

Chandra and Toueg [CT96] introduced the concept of unreliable failure detectors and how they can be used to solve consensus in fault-tolerant asynchronous distributed systems. They characterize unreliable failure detectors in terms of two properties: *completeness* and *accuracy*.

In general terms, the completeness property requires that every faulty process should be suspected, while the accuracy property restricts the false suspicions (mistakes) that a failure detector can make. In other words, completeness specifies the capacity of a failure detector to suspect a failure, and accuracy limits the mistakes of a failure detector while it is suspecting.

Chandra and Toueg [CT96] identified eight classes of failure detectors (see Figure 2.5) that combine properties of completeness and accuracy.

2.4.1.1 Completeness Properties

The first property that a failure detector must satisfy is the integrity property, which means that if a process is faulty, then the failure detector should suspect it. The *completeness* property can be classified in:

- *Strong Completeness.* Every faulty process is eventually and permanently suspected by every non-faulty process.
- *Weak Completeness.* Weak completeness: every faulty process is eventually and permanently suspected by some non-faulty process.

Completeness by itself is not very useful. Given that strong completeness can be trivially satisfied by forcing every process to suspect every other process in the system permanently. However, such a failure detector is useless since it provides no information about failures.

2.4.1.2 Accuracy Properties

The second kind of property is accuracy, which means that, if a process is suspected, then it has failed. In general terms, this property restricts the mistakes that a failure detector can make. The *accuracy* property can be classified in:

- *Strong Accuracy.* No process is suspected before it fails (also known as *Perpetual Strong Accuracy*).
- *Weak Accuracy.* Some correct process is never suspected (also known as *Perpetual Weak Accuracy*).
- *Eventual Strong Accuracy.* There is a time after which correct processes are not suspected by any correct process.
- *Eventual Weak Accuracy.* There is a time after which some correct process is never suspected by any correct process.

Note that the failure detectors with eventual accuracy may suspect every process at one time or another, while failure detectors with perpetual accuracy require that at least one correct process is never suspected.

2.4.2 Failure Detector Classes

Failure detectors can be classified based on the properties that they satisfy (*completeness* and *accuracy*). In Figure 2.5, we show the hierarchy of failure detector classes established by Chandra and Toueg [CT96].

SYNCHRONOUS ← ASYNCHRONOUS

COMPLETENESS ↑	ACCURACY ←			
	STRONG	WEAK	EVENTUAL STRONG	EVENTUAL WEAK
STRONG	*Perfect* P	*Strong* S	*Eventually Perfect* $\diamond P$	*Eventually Strong* $\diamond S$
WEAK	*Quasi-Perfect* Q	*Weak* W	*Eventually Quasi-Perfect* $\diamond Q$	*Eventually Weak* $\diamond W$

Figure 2.5: Classification of Failure Detectors

Additionally, in [CT96] it is stated that any failure detector that meets *weak completeness* can be transformed into a failure detector that meets *strong completeness* with the same accuracy property.

We can understand the hierarchy of such classes of failure detectors as follows:

- We assume that class $\mathcal{A}$ *is stronger than* class $\mathcal{B}$ if the information about the failure processes provided by the failure detector of class $\mathcal{A}$ also includes the information provided by the failure detector of class $\mathcal{B}$.
- We focus on the *eventual weak accuracy* property. We can see from the definition that there is a time after which at least *one correct process is never suspected by any correct process*, while in the *eventual strong accuracy* property *every correct process eventually is not suspected.*

Often, the study of failure detectors focuses on those that satisfy the *strong completeness* property (first row of Figure 2.5). Moreover, Chandra and Toueg [CHT96] showed that $\diamond\mathcal{S}$ is the weakest class of failure detectors allowing to solve the distributed consensus problem in an asynchronous system with a majority of correct processes.

Remark.

- The *Perfect* failure detector ($\mathcal{P}$) is *reliable* in the sense that every suspected process is a faulty process.
- The *Eventually Perfect* failure detector ($\diamond\mathcal{P}$) is *unreliable* since it can make mistakes, even though eventually it will make no mistake.
- Guerraoui in [Gue00, GR04] showed that algorithms devised on the approach of *unreliable failure detection are indulgent algorithms*, meaning that they never violate their safety properties.

2.4.3 The Omega Failure Detector

In addition to the previous classes, Chandra, Hadzilacos and Toueg [CHT96] introduced another failure detector class, called *Omega* (Ω).

The *Omega* failure detector was presented as the weakest failure detector for solving consensus with a majority of correct processes [CHT96]. From some point of time, Omega allows providing an agreement on a correct process among all processes that are non-faulty in a system.

Chandra, Hadzilacos and Toueg [CHT96] defined the Omega failure detector for the crash failure model. Basically, the output of the failure detector module of Omega at a process p is a single process q, that p currently considers being correct (we say that p trusts q).

Formally, the Omega failure detector (Ω) has been defined as follows: *there is a time after which every correct process always trusts the same correct process.* Observe that the definition, made for the crash failure model, does not state anything about incorrect processes, which are allowed to disagree at any time with correct processes.

The *Eventually Strong* ($\diamond\mathcal{S}$) failure detector class would be enough to solve consensus. However, $\diamond\mathcal{P}$ is a more natural failure detector class, in the sense that it ensures that eventually, all correct processes have a suspected list that exactly contains incorrect processes, providing a higher degree of accuracy than $\diamond\mathcal{S}$. On the other hand, the Omega failure detector (Ω) provides a leader election mechanism, which supports leader-based consensus protocols.

Some interesting observations about the *Omega* failure detector, which make it attractive for our work, are:

- The *Omega* failure detector is *equivalent* to the *Eventually Strong* failure detector ($\Omega \equiv \diamond\mathcal{S}$) [CHT96, LAA13].
- The *Omega* failure detector guarantees that, eventually, all the correct processes agree permanently on a common correct process, i.e., provides an *eventual leader election functionality* [MOZ05, ADFT08, MLJ09, SCL$^+$11, LMA11].
- The *Omega* failure detector is a *trust-based* failure detector, whereas the previous ones are *suspicion based* failure detectors.
- The *Omega* failure detector allows designing indulgent protocols [Gue00, GR04].

2.4.4 The Notion of Failure Detector Reduction

Chandra et al. [CHT96] introduced the relation "*weaker than*" to compare failure detectors. Later, the notion of failure detector reduction ("*weaker than*") has been improved in [JT08].

A failure detector $\mathcal{F}_{\mathcal{D}1}$ is "*weaker than*" another failure detector $\mathcal{F}_{\mathcal{D}2}$, if there is an asynchronous algorithm which can emulate $\mathcal{F}_{\mathcal{D}1}$ by using $\mathcal{F}_{\mathcal{D}2}$. In other words, $\mathcal{F}_{\mathcal{D}2}$ is reducible to $\mathcal{F}_{\mathcal{D}1}$ by a reduction algorithm, and it is denoted as $\mathcal{F}_{\mathcal{D}1} \preceq \mathcal{F}_{\mathcal{D}2}$,

Other definitions are presented in [CHT96, JT08] as follows:

- *Equivalent.* If $\mathcal{F}_{\mathcal{D}1} \preceq \mathcal{F}_{\mathcal{D}2}$ and $\mathcal{F}_{\mathcal{D}2} \preceq \mathcal{F}_{\mathcal{D}1}$, then $\mathcal{F}_{\mathcal{D}1} \equiv \mathcal{F}_{\mathcal{D}2}$.
- *Strictly-Weaker.* If $\mathcal{F}_{\mathcal{D}1} \preceq \mathcal{F}_{\mathcal{D}2}$ and $\mathcal{F}_{\mathcal{D}2} \npreceq \mathcal{F}_{\mathcal{D}1}$, then $\mathcal{F}_{\mathcal{D}1} \prec \mathcal{F}_{\mathcal{D}2}$.

Some interesting results concerning the reducibility:

- A reduction algorithm is assumed to be asynchronous, and, as a result, it does not implement any function in terms of failure detection.
- A reduction algorithm often implements a distributed variable where the result is an emulated failure detector.
- A reduction algorithm depends on the system model, i.e., the relation "*weaker than*" is only valid on the same model.

Therefore, the ability to compare failure detectors motivate us to chase the *weakest failure detector* to solve a given problem $\mathcal{P}$. Formally, a failure detector $\mathcal{F}_{\mathcal{D}}$ is the weakest failure detector to solve $\mathcal{P}$ if:

- There is an algorithm that solves $\mathcal{P}$ using $\mathcal{F}_{\mathcal{D}}$.
- $\forall$ $\mathcal{F}_{\mathcal{D}1}$' such that there is an algorithm that solves $\mathcal{P}$ using $\mathcal{F}_{\mathcal{D}}{}'_1 : \mathcal{F}_{\mathcal{D}1} \preceq \mathcal{F}_{\mathcal{D}}{}'_1$

In reference to the failure detector classes (see Figure 2.5), we can deduce the following:

- $\mathcal{P}$ can simulate any class of failure detectors. $\mathcal{S}$ and $\diamond\mathcal{P}$ can both simulate $\diamond\mathcal{S}$ but cannot simulate $\mathcal{P}$. $\diamond\mathcal{S}$ cannot simulate any of the other classes. In consequence, $\diamond\mathcal{S}$ is the weakest class of failure detectors.
- $\diamond\mathcal{S}$ is the weakest failure detector for solving distributed consensus problem with a majority of correct processes [CHT96].

The chase of the weakest failure detector for many problems has been investigated in [CHT96, CT96, ATD99, DFG+04, JT08]

2.4.5 Approaches to Implementing Failure Detectors

There are different approaches to implement a failure detector. Each process monitors the other processes in the system through a local failure detector module. In general, the most commonly used ways to make such supervision are:

- **Polled Input/Output.** When a process *p* monitors another process *q* through polling (also known as Query/Reply), *p* sends a query message to *q* and stays waiting for an answer to this message from *q*. If *p* does not receive an answer after a given time, *p* will suspect *q*, i.e., *p* will consider *q* as faulty. Different failure detectors based on polling have been proposed [LAF99, LFA00, LFA04].
- **Heartbeat.** In a failure detector, a heartbeat protocol is frequently used to monitor the availability of processes [ACT97, ACT99]. Every supervised process *q* sends a heartbeat message periodically to monitoring processes to notify them that it is still alive. If the monitoring processes do not receive the expected message from *q* after a given time, then they will suspect *q*. In both cases, the waiting time depends on the timing of the system model. In a synchronous system, that time is known and can be estimated previously. However, in a partially synchronous system, a mechanism of auto-adjust the time-out is necessary.
- **Communication Patterns.** It is an essential issue when looking for efficiency in communication, given that the failure detector implementations can follow different communication patterns [LAF99, ADFT04, FLCR16]. As an example, an all-to-all communication pattern presents advantages, such as responsiveness, but lacks efficiency regarding communication. A linear communication pattern is desirable to reach a higher degree of efficiency. In Chapter 4, we address the effect of the communication pattern on communication efficiency.

2.5 Building Blocks for Fault-Tolerant Applications

Today, most fault-tolerant distributed applications can be built in a modular way. The building blocks have become essential for the protocols solving agreement problems, such as consensus algorithms (see Figure 2.6).

As mentioned earlier, a failure detector allows encapsulating the synchrony assumptions. Therefore, fault-tolerance distributed applications can be designed as if they were executed in an asynchronous system. An alternative and elegant approach to circumvent the FLP impossibility of consensus in asynchronous systems was proposed in [CT96].

We take the work of Cortiñas et al. [CFGA+12] as a reference since it shows how failure detectors and consensus can be used in a modular way to build fault-tolerant distributed applications. More precisely, the building blocks are used to solve *Secure Multiparty Computation* [Yao82]

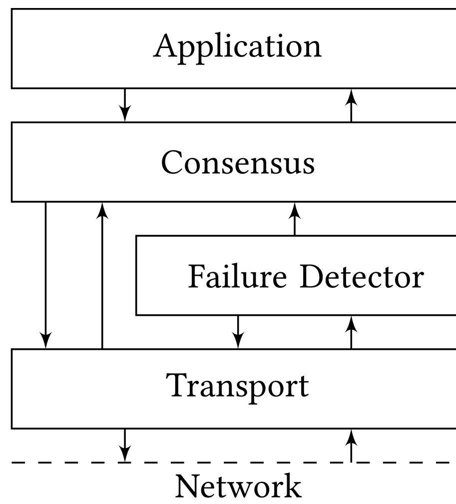


Figure 2.6: Building blocks: Consensus and Failure Detectors

in the Byzantine failure model [LSP82]. In the aforementioned work, they augmented an asynchronous model with an unreliable failure detector. In this way, failure detectors offer a modular approach that allows other applications (as consensus) to use them as a building block (see Figure 2.6).

3 A Performance Study of Consensus Algorithms

"If two people always agree on everything, I can assure that one of the two is thinking for both."
— Sigmund Freud[3]

In this chapter, we present a practical comparison of two well-known distributed consensus algorithms, with the idea of extending and improving the work presented in [CFGA+12]. We use the *TrustedPals framework* as a case study of applicability since it allows us to study the problem of uniform consensus in a practical way. We have performed simulations to compare two consensus algorithms under different scenarios of failures, using modeling techniques and simulation tools. The results show better behavior and convergence in the improvement, which is based on a mechanism of leader election. We believe that the proposed approach will facilitate the development of distributed applications in highly dynamic and heterogeneous environments.

Outline. The chapter is organized as follows. In Section 3.1, we present the context of the study. An overview of consensus algorithms is presented in Section 3.2. The applicability of the consensus problem is shown in Section 3.3, through the case study. Section 3.4 presents a practical comparison of both consensus algorithms. The experimental results and analysis obtained are shown in Section 3.5. Finally, the summary of the chapter is given in Section 3.6.

3.1 The Study Context

Dependable distributed systems are systems where reliability and availability are critical aspects, both concerning communication and computation. Distributed systems are prone to failures, either due to errors in the network infrastructure or devices or caused by attacks from an adversary. As a consequence, a part of the entire system can behave inconsistently and unpredictably, which affects the reliability of the system. For this reason, providing a high degree of fault-tolerance and security to the system is a crucial aspect of the design and development of distributed applications in scenarios where a potentially large set of interconnected devices cooperate to achieve an aim.

In asynchronous distributed systems prone to crash failures, many problems require all the processes to reach agreement on a decision value. Therefore, providing dependability [Lap87,

[3] Austrian scholar, psychiatrist (1856–1939).

Lap92] to fault-tolerant distributed systems requires solving agreement problems, such as *the consensus problem*. Additionally, the consensus problem can be used as a building block to solve some other problems related to distributed systems.

With the idea of extending and improving the work presented in [CFGA+12], we study and present a new proposal of architecture. We investigate the performance of two well-known consensus algorithms, using as a case study a smart card-based framework that allows implementing security policies in distributed systems, known as TrustedPals framework [FFP+06, CFGA+12].

We use the TrustedPals framework as a case study of the applicability of a solution to the uniform consensus problem. The distributed consensus algorithms of the study are *Chandra-Toueg* [CT96] and *Paxos* [Lam98, PLL00, Lam01]. The architecture of the TrustedPals framework proposed in [CFGA+12] uses the Chandra-Toueg consensus algorithm adapted to the omission failure model.

In this chapter, we propose the use of the Paxos consensus algorithm as an alternative to extend the applicability of the framework to the crash-recovery failure model as well. We perform a comparison of the performance of both distributed consensus algorithms through simulation under different scenarios of failures. The results obtained show a better behavior and performance of our proposal, which relies on an eventual leader election mechanism [FCL13, FCL14b, CRA14].

3.2 An Overview of the Considered Consensus Algorithms

Protocols that solve agreement problems are essential building blocks for fault-tolerant distributed applications. Many consensus protocols have been proposed in recent decades. Consensus protocols are the foundation for the state machine replication approach in distributed computing, as suggested by Lamport [Lam78] and surveyed by Schneider [Sch90]. Nowadays, Paxos is the most popular consensus protocol.

State machine replication is the method widely used to implement a fault-tolerant service [Sch90, Lam96]. State machine replication aims to make a group of distributed processes (also known as replicas) to execute the same commands in the appropriate order, regardless of failures. The different replicas may operate independently and asynchronously.

Two consensus algorithms are studied, Chandra-Toueg [CT96] and Paxos [Lam98, PLL00, Lam01], since they have many characteristics in common, and therefore it is relatively easy to define a meaningful comparison. Both algorithms are described in more detail below.

3.2.1 Chandra-Toueg's Algorithm

The Chandra–Toueg consensus algorithm was published in 1996 [CT96], it was proposed to solve the consensus problem in asynchronous systems with crash failures. The Chandra–Toueg

consensus algorithm uses an unreliable failure detector, denominated *Eventually Strong Failure Detector*. It is the weakest class allowing to solve Consensus [CHT96].

The *Eventually Strong Failure Detector*, also known as $\diamond\mathcal{S}$, satisfies the following completeness and accuracy properties (defined in [CT96]):

(i) *Strong Completeness.* Eventually, every process that crashes is permanently suspected by every correct process, and

(ii) *Eventual Weak Accuracy.* There is a time after which some correct process is never suspected by any correct process.

The algorithm requires reliable communication and tolerates crash failures. Moreover, it assumes that the number of faulty processes is less than $n/2$, where n is the total number of processes. Therefore, the Chandra–Toueg consensus algorithm requires *a majority of correct processes in the system.*

A *rotating coordinator mechanism* allows to Chandra–Toueg consensus algorithm to ensure their termination (*liveness property*). This means that some rounds are under the control of the same process, each round being coordinated by a particular process. The identity of the process that coordinates a specific round is predetermined from the value of a simple mathematical equation.

3.2.2 Lamport's Paxos Algorithm

Paxos algorithm was proposed by Lamport [Lam98, PLL00, Lam01]. Since then, Paxos has become one of the most studied and used consensus protocols, and many variants and adaptations have been proposed, e.g., *Disk-Paxos*, *Cheap-Paxos*, *Fast-Paxos*, *Byzantine-Paxos* ...). Paxos provides flexibility, allows the detection of failures in the crash-recovery failure model, and facilitates the development of distributed applications in highly dynamic and heterogeneous environments.

A *leader election mechanism* allows Paxos to ensure termination (*liveness property*). Interestingly, in some scenarios, Paxos cannot guarantee progress (*liveness property*), but it always preserves the *safety property* of consensus, despite asynchrony and process failures. Paxos guarantees progress when the leader is unique and can communicate with a majority of correct processes. This could imply additional requirements, e.g., *a majority of correct processes in the system* (see Chapter 4).

Interestingly, the leader election mechanism required by *Paxos* can be provided by a well-known failure detector proposed by Chandra, Hadzilacos and Toueg, *Omega* (Ω) [CHT96]. This failure detector (Ω) provides an eventual agreement on a common leader among all non-faulty processes in a system. It is noteworthy that Ω is the weakest failure detector for solving consensus in crash scenarios. Observe that Ω can be trivially obtained from $\diamond\mathcal{P}$, e.g., by choosing as leader the non-suspected process with the lowest identifier. Although initially proposed for crash-prone systems, more recently, *Omega* has been studied in crash-recovery systems as well [MLJ09].

3.2.2.1 Variants of Paxos Algorithms

There are many variants of the classic Paxos distributed consensus algorithm. These different options focus on making the classic Paxos algorithm more efficient in more difficult environments. In this section, we briefly introduce some of them.

The *Disk Paxos* [GL00] algorithm implements a reliable distributed system across a network of processors and disks. It uses a consensus algorithm to agree on the transitions of a replicated state machine, where progress can be guaranteed as long as most disks are available. Note that the algorithm maintains consistency in the presence of arbitrary non-byzantine faults.

The *Cheap Paxos* [LM04] algorithm can guarantee a liveness property in the system, only under the additional assumption (a set of $f + 1$ non-faulty processors). Hence, substitute processors (f) are only used to handle the failed main process , where f is the tolerance number to concurrent faults. These secondary processors take part in the reconfiguration of the system to eliminate the processor failure, after which they can remain inactive until another main processor fails.

The *Fast Paxos* [Lam06a] algorithm allows agreement to occur in two message delays, provided that there is no collision, and even with a collision agreement can be guaranteed in three message delays. Besides, it can achieve any desired degree of fault-tolerance using as few processes as possible.

The *Fast Byzantine (FaB)* [MA06] algorithm was the first Byzantine Paxos protocol. It only requires two communication steps to reach consensus in the common case. Confirming a conjecture by Lamport [Lam06b], latency reduction implies an additional price: *FaB* requires $5f + 1$ acceptors to tolerate f Byzantine acceptors, instead of the $3f + 1$ needed by previous protocols.

The *Vertical Paxos* [LMZ09] algorithm allows reconfiguration while the replicated state machine is active and decides the commands. Vertical Paxos uses a secondary "master" to manage reconfiguration operations, which determine the set of acceptors and the leader for every configuration.

The *Egalitarian Paxos (EPaxos)* [MAK13] algorithm is proposed to solve the single leader bottleneck. At afford to every replica receive values from the client, and these values can be with a message delay, only if there is no dependence between commands. Further, it allows distributing the load to all replicas evenly.

The *Raft* [OO14] algorithm is designed to be easy to understand. It is equivalent to Paxos in fault-tolerance and performance. The main difference is that the Raft algorithm decomposes into relatively independent sub-problems, and it is designed in a practical way to build distributed applications.

3.2.3 Some Points of Comparison between both Algorithms

We summarize some common points of Chandra–Toueg and Paxos consensus algorithms. Both algorithms operate in an asynchronous system model with classes of equivalent failure detectors. As an additional requirement for correctness, they require most of their processes to be correct. Also, they have a similar structure in their execution.

3.2.3.1 Similarities in Execution

Both consensus algorithms (Chandra-Toueg and Paxos) use a similar structure in their execution, i.e., a sequence of rounds in which each can be represented through four phases, as described below:

- **Phase 1.** In the Chandra-Toueg algorithm, all processes send their *estimated value* to the *coordinator* process. In the case of the Paxos algorithm, the *leader* process sends a *preparation message* to the other processes.

- **Phase 2.** The coordinator (Chandra-Toueg) or leader (Paxos) process waits for messages from a majority of processes. Note that some messages might not arrive in case of failure (crash, omission, or crash-recovery). Further, a process can reject the proposal if it suspects that the coordinator process has failed (Chandra-Toueg) or if an earlier proposal has been accepted for the same ballot number (Paxos).

- **Phase 3.** When the coordinator (Chandra-Toueg) or leader (Paxos) process receives a majority of positive responses, an acceptance message is issued with the value to agree on, which is based on the responses in the previous phase. A process replies with an acceptance message, only in case it has not suspected that the coordinator has failed (Chandra-Toueg) or an earlier proposal has not been accepted for the same ballot number (Paxos). Otherwise, it rejects the proposal.

- **Phase 4.** If a majority of processes accept the proposal, then the decision reached is transmitted to all processes. Therefore, the coordinator (Chandra-Toueg) or leader (Paxos) process sends a message with the value decided to all processes in the system.

Each round has a coordinator/leader process that tries to impose a decision on all processes, and if this fails, a new round is started with a new coordinator/leader. Observe that both algorithms can execute an indeterminate sequence of rounds, and only finish when agreement is reached.

3.2.4 Both Algorithms Solve Fault-Tolerant Agreement

Both protocols solve agreement problems, and they are essential in the modular construction (as building blocks) of fault-tolerant distributed systems. The consensus algorithm of Chandra-Toueg has some similarities with Paxos. In the Chandra-Toueg algorithm, it also uses leader-driven rounds (coordinator), but rounds are performed sequentially, while in the Paxos algorithm, a leader can start a new round at any time allowing various leaders to coexist.

They are two well-known asynchronous consensus algorithms. As mentioned earlier, a leading process attempts to reach an agreement, and if it fails, another leader is elected, and retries reach the agreement. Both algorithms structure their executions into rounds. In each round, a process (denoted as a *coordinator* in Chandra-Toueg and as a *leader* in Paxos) tries to impose a decision. The algorithms differ in how they choose the coordinator/leader process for the next round. In Chandra-Toueg, the coordinator role rotates among all processes, whereas in Paxos, the leader is directly selected from an uncoordinated manner (wherein a process may be the leader in consecutive rounds).

Note that the Chandra-Toueg algorithm considers a distributed setting that does not allow process restarts (crash and recovery) and faulty communication links. However, it can be modified to work with the loss of messages. The Paxos algorithm tolerates process restarts (crash and recovery) and faulty communication links, making Paxos more suitable in the use of more practical applications. Additionally, they have a high capacity to tolerate failures, being able to tolerate up to $\frac{n}{2} - 1$ concurrent failures in a system with n processes.

The importance of these consensus algorithms lies in their robustness, both offering the ability to preserve the *safety* property (i.e., always acting according to its specification) despite asynchrony in the system. Both algorithms are considered indulgent consensus algorithms [Gue00, GR04], i.e., they are indulgent toward their failure detector in the sense that they never violate the safety property of consensus, no matter how the underlying failure detector behaves.

As another plus point, Paxos distinguishes three roles among the entities that participate in the consensus, namely *proposer*, *acceptor*, and *learner*, providing the system with a more fine-grained role assignment compared to the Chandra-Toueg algorithm. Furthermore, as previously mentioned, the Paxos algorithm can deal with crashed and crash-recovered processes as well as with message omissions in a more natural way.

3.3 Case Study: Improving the TrustedPals Framework

We seek to extend and improve the work presented in [CFGA+12]. Thus, although the proposed solution is suitable in the security context, it presents some drawbacks that should be improved in order to be applied in weaker scenarios. For this reason, we use the TrustedPals framework as a case study, since it allows us to study the distributed consensus problem and propose improvements.

3.3.1 Solving Yao's Millionaire's Problem

The *Yao's Millionaire's problem* is used as a generic definition of the Secure Multiparty Computation problem [Yao82]. A solution to this general security problem was presented in [FFP+06], namely TrustedPals, in which Secure Multiparty Computation is implemented by using smart cards.

3.3.1.1 Secure Multiparty Computation Problem

The *Yao's Millionaire's problem* was introduced by Andrew Yao [Yao82], a computer scientist and computational theorist. He presented this problem through the following sentence:

"[...] two millionaires, Alice and Bob, who are interested in knowing which of them is richer without revealing their actual wealth."

The difficulty of this problem lies in developing a joint computation and communication protocol to be executed among multiple distrusted network nodes without disclosing any private information. Such a protocol is called *Secure Multiparty Computation* [Yao82], and it has been an active research area in cryptography for more than thirty years.

The *Secure Multiparty Computation* (SMC) is a subfield of cryptography with the goal to create methods for parties to compute a function over their inputs jointly, and keeping those inputs private. Given that encryption alone can not provide adequate protection to implement applications that satisfy the above problem. Because the encrypted data must be decrypted in the receiver for processing, and the raw data will be vulnerable. Then, *Trusted Computing* [And03] can solve that problem by running the software in a secure memory space of the client machine equipped with a cryptographic coprocessor. Roughly speaking, solving SMC among processes is achieved by having security modules jointly simulate a *Trusted Third Party* [Küp13].

In this way, SMC has transformed a very general security problem, i.e., it can be used to solve various real-life issues such as distributed voting, private bidding, and online auctions, sharing of signature or decryption functions. Besides, SMC is increasingly used in diverse fields, from data mining to computer vision. Unfortunately, solving SMC is very expensive in terms of communication (*number of messages*), resilience to failure (*amount of redundancy*), and time (*number of synchronous rounds*).

3.3.1.2 TrustedPals Framework

The TrustedPals framework [FFP+06, CFGA+12] (indistinctly, we will call it TrustedPals) is a smart card-based implementation that allows reaching more efficient solutions to the SMC problem. Initially, TrustedPals [FFP+06] assumed a synchronous network setting and allowed to reduce SMC to the problem of fault-tolerant consensus among smart cards. Conceptually,

TrustedPals considers a distributed system in which processes are locally equipped with tamper-proof security modules. In practice, processes are implemented as a *Java* desktop application, and security modules are realized using *Java Card Technology* [Che00] enabled smart cards.

Cortiñas et al. [CFGA+12] show how to reduce the SMC problem to the consensus problem and how to solve the latter in a Byzantine failure model by using an *Eventually Perfect Failure Detector* ($\diamond\mathcal{P}$) adapted to omission environments, and all into TrustedPals. More precisely, it makes use of a tamper-proof smart card-based secure platform (presented in [FFP+06]) to solve the consensus problem in a partially synchronous system prone to Byzantine failures, in which the malicious behavior can be reduced to a more benign model of omission failures. The solution of Cortiñas et al. [CFGA+12] is based on a uniform consensus and *all-to-all* communication pattern, which helps to preserve security, but it is rather inefficient in terms of the number of exchanged messages. For such reason, Soraluze et al. [SCL+11] propose a more efficient solution that tries to avoid redundant communication.

We explore how to make TrustedPals applicable in environments with less synchrony and show how it can be used to solve asynchronous SMC. Our study is based on the work realized by Cortiñas et al. [CFGA+12], which we denote "current architecture" of TrustedPals. In the current architecture of TrustedPals, the Chandra-Toueg consensus algorithm is combined with a $\diamond\mathcal{P}_{om}$ failure detector class. Note that $\diamond\mathcal{P}_{om}$ is a $\diamond\mathcal{P}$ failure detector class from [CT96] adapted to the general omission failure model.

Another point to note, the solution in [CFGA+12] tolerates some message omissions and process crashes. Nevertheless, that tolerance is limited mainly due to the use of the consensus algorithm proposed by Chandra and Toueg [CT96] which, besides requiring a majority of correct processes, it is rather sensitive to message omissions. Initially, the algorithm was designed assuming reliable communication.

3.3.2 A Proposal for Improving TrustedPals

In search of an improvement of the approach presented in Cortiñas et al. [CFGA+12], we present how to weaken the assumptions of the system to cope with more dynamic environments, and thus, to extend the applicability of TrustedPals to the crash–recovery failure model. Additionally, we are looking for more efficient solutions in terms of latency and quantity of messages exchanged.

As an initial step, we propose replacing the Chandra-Toueg consensus algorithm used in [CFGA+12] by the Paxos consensus algorithm augmented with an *Omega* failure detector class (Ω), in order to achieve less restrictive solutions in omission environments such as crash-recovery scenarios.

In the following questions, we will explore what hypothesis we want to demonstrate:

- *What are we looking for?* A practical comparison of two well-known distributed consensus algorithms, with the purpose of extending and improving the work presented in [CFGA+12]. Consequently, the study of efficient solutions to solve the consensus problem in partially synchronous distributed systems under different failure models.

- *How?* Weakening the assumptions of the model system to deal with more dynamic environments and, therefore, extend the applicability of TrustedPals to crash-recovery.
- *Why?* We think that using the Paxos distributed consensus algorithm, enriched with an Omega failure detector in a crash-recovery failure model, improves the TrustedPals framework substantially.

3.3.3 System Model and Assumptions

We consider a distributed system composed of a finite set of processes, such processes communicate only through the exchange of messages. Additionally, we do not consider the use of shared memory.

The communication graph is fully connected, i.e., there is a bidirectional communication link between every pair of processes in the system. Communication links are reliable, i.e., no message from a non-faulty process is dropped, duplicated, or modified, and the links do not generate any message.

We assume a majority of correct processes in the system, and this is a requirement for both consensus algorithms used in this study.

Regarding *timing assumptions*, we consider a *partial synchrony model* proposed in [CT96], which can be summarized as:

- We define as: Time $\implies \Delta$ (message delay) + ϕ (processing time)
  - Δ, upper bound on message delay (*unknown*)
  - ϕ, upper bound on processing time (*unknown*)
- We assume that: there exists a Global Stabilization Time (GST), such as:
  - Before GST, the system presents an *asynchronous* behavior, i.e., bounds do not hold ($\Delta + \phi$)
  - After GST, the system presents a *synchronous* behavior, i.e., bounds hold ($\Delta + \phi$)
  - GST is *unknown*

Regarding the *failure models*, we consider several failure types that could suffer a process through the use of a gradual approach. Initially, we only consider the general omission failure model, in which processes may crash by prematurely halting, or suffer omissions of messages either by sending (*send omission*) or receiving (*receive omission*). The general omission failure model was proposed by Hadzilacos [HT93] and later generalized by Perry and Toueg [PT86]. Additionally, we also consider the crash-recovery failure model in which a process may fail and later recover [MLJ09].

Broadly speaking and except for the crash-recovery failure model, we consider the same system model as was presented in the work of Cortiñas et al. [CFGA+12].

3.3.4 Architecture of TrustedPals

In Figure 3.1 (left side), we present the current architecture of TrustedPals, which is composed of the following three modules:

- The top module presents the user application, SMC in [CFGA+12], which makes use of the consensus service at the lower level.
- The core module, namely TrustedPals, provides the consensus service. It is composed of a consensus algorithm adapted from the original algorithm presented in [CT96], combined with an Eventually Perfect ($\diamond\mathcal{P}$) failure detector adapted to the general omission failure model ($\diamond\mathcal{P}_{om}$). This failure detector provides information about *well-connected* processes, i.e., processes that can actively participate in the consensus. Together with this module, there is a smart card-based secure platform also named TrustedPals, which allows reducing Byzantine failures to more benign process crashes or message omissions.
- Finally, the bottom module represents the distributed system in which the application has to be executed.

Although the proposed solution is suitable in the security context presented of [CFGA+12], it presents some drawbacks that could be improved in order to be applied in other scenarios. For example, it does not consider processes that crash and later recover. Also, the consensus algorithm requires a high degree of reliability on communication, i.e., non-omissive processes and reliable channels.

3.3.5 Obtaining Ω_{Om} From a Simple Reduction of $\diamond\mathcal{P}_{om}$

As introduced in 2.4, a failure detector encapsulates the timing assumptions for a specific system. It can be defined as an abstract module that provides information about the current state of processes in a distributed system, and the information obtained is not necessarily accurate.

By definition, an *Eventually Perfect* failure detector ($\diamond\mathcal{P}$) satisfies the following properties (see Chapter 2.4):

- *Strong Completeness:* There is a time after which every correct process permanently suspects *every* failed process.
- *Eventual Strong Accuracy:* There is a time after which every correct process is never suspected by any correct process.

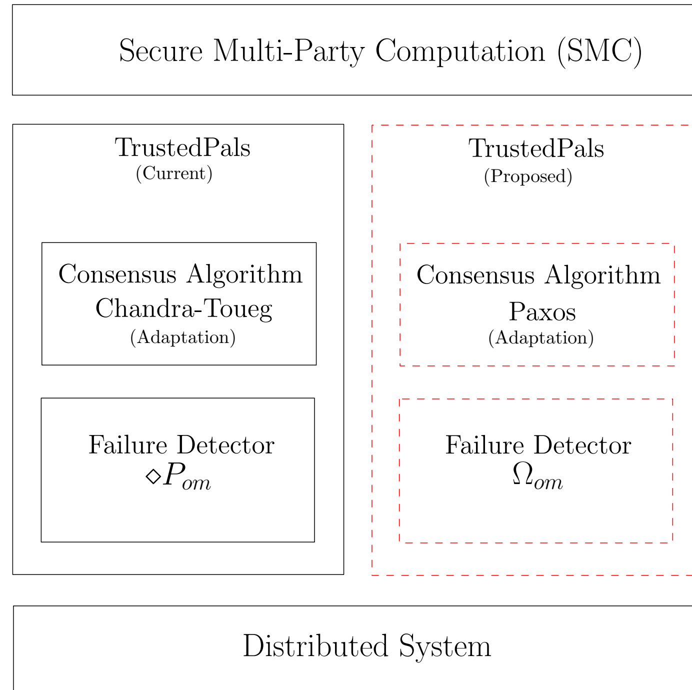


Figure 3.1: Current and Proposed Architecture for TrustedPals

Can we obtain an eventual leader election mechanism fault-tolerant in the omission failure model, from an eventually perfect failure detector?

As an initial approximation, we implement the Ω_{Om} failure detector by using the $\diamond\mathcal{P}_{om}$ failure detector presented in [CFGA+12]. Then, it requires adapting the definition of Ω_{Om} to the *connectedness* properties presented in [CFGA+12]. This way, since in Paxos, the leader must be able to send and receive messages from a majority of processes, our specification for the omission model is "*Eventually every in-connected process trusts forever the same well-connected process (i.e., a process that is both in-connected and out-connected)*". Remember that we assumed a majority of correct (i.e., *well-connected*) processes in the system.

Following the previous definition, we transform $\diamond\mathcal{P}_{om}$ into Ω_{Om} by electing as leader the process with the lowest identifier among the processes considered *in-connected* and *out-connected* by $\diamond\mathcal{P}_{om}$. This requires a small adaptation of the $\diamond\mathcal{P}_{om}$ algorithm in order to calculate the set of *in-connected* and *out-connected* processes, rather than just the set of *out-connected* processes.

This initial approximation to the implementation of Ω_{Om} presents two advantages:

(i) As in Cortiñas et al. [CFGA+12], the use of a periodical all-to-all communication pattern, inherent to the implementation of $\diamond\mathcal{P}_{om}$, allows tolerating malicious attacks.

(ii) From a practical point of view, it allows making a fair comparison of the two consensus algorithms, since the cost of implementation of the failure detector module in both cases is the same.

3.3.6 A Novel Design of Architecture for TrustedPals

In Figure 3.1 (right side), we present a novel proposal for the architecture of the study. Note that it keeps the modular approach of the previous one, but with two main differences:

- The Chandra-Toueg consensus algorithm is replaced by the Paxos algorithm [Lam98, PLL00, Lam01].
- The failure detector considered now is an adaptation of [CHT96] for the general omission failure model, denoted by Ω_{Om}.

The combination of those two new elements in the architecture provides the system with several interesting features:

(i) On the one side, the Paxos algorithm allows reaching consensus tolerating a high degree of omissive behavior (loss of messages at processes or communication links). Also, the Paxos algorithm allows coping with processes that crash and later recover.

(ii) On the other hand, the Omega failure detector tolerant of omissive failures, Ω_{Om} , provides a Paxos algorithm with the eventually stable leader required to guarantee termination.

The proposed architecture requires an Ω failure detector class for the general omission and crash-recovery failure models, class which we denote Ω_{Om} for the case of omission failure (that we presented in the previous section).

3.4 A Practical Comparison of both Consensus Algorithms

When studying distributed consensus algorithms, it is essential to consider that the protocols that solve agreement problems are used as building blocks of fault-tolerant distributed applications. In this section, we show the results of a practical comparison of both architectures for TrustedPals (see Figure 3.1).

In the literature, many studies are concentrating on the analysis in order to prove the correctness of consensus algorithms. However, there are very few studies focused on the different failure models and how they affect the performance of consensus.

We have performed simulations to compare these two consensus algorithms under different failure scenarios. Besides crash failures, we also consider scenarios with message omissions and where processes also may suffer crash-recovery failures. Our results reveal that Paxos is more efficient than Chandra-Toueg's consensus algorithm if the first process that coordinates or leads a round, suffers a failure, while both algorithms work similarly when there are no failures.

3.4.1 Related Studies

Many protocols have been published, but little has been done to analyze their performance, especially the performance of their fault-tolerance mechanisms. Besides, few studies use metrics to analyze their performance, such as the time complexity (*number of communication steps*) or the message complexity (*number of messages exchanged*).

Among the latter are found the works of Hayashibara et al. [HUSK02] and Urban et al. [UHSK04]. Both studies present a comparison in terms of latency and throughput of the classical (i.e., non-adapted) Chandra-Toueg and Paxos, by simulating them in scenarios without failures and with just crash failures (i.e., without regard to omissions and crash-recovery failures). For its part, Coccoli, Urban and Bondavalli [CUB02] analyze the latency of a consensus algorithm augmented with a failure detector simulated in a stochastic network (by conducting experiments on a cluster). They consider performance metrics related to the quality of service (*QoS*) of the failure detector based on the previous work of Chen, Toueg and Aguilera [CTA02]. Sergent, Defago and Schiper [SDS01] study the impact of the failure detection mechanism on the performance of the consensus algorithm, again in crash scenarios. Lastly, Borran et al. [BHSS12] compare analytically different round-based consensus algorithms, in a partially synchronous system that alternates between synchronous and asynchronous periods.

3.4.2 Simulations

We implement our approaches by using JBotSim [Cas15]. Using this simulation software, we have implemented adaptations of the consensus algorithms (Chandra-Toueg and Paxos consensus algorithms). To obtain the minimum distortion in the simulation of these algorithms, we developed a failure detector through the reduction method (from $\diamond\mathcal{P}_{om}$ to Ω_{Om}). Additionally, we implement a method to measure the performance of each algorithm under various failure scenarios.

3.4.2.1 Simulation Tool: *The JBotSim Library*

JBotSim is a Java library for the simulation of distributed algorithms, with a style of programming event-driven based, and its main advantage versus other alternatives is its abstraction regarding the characteristics of the underlying network, thereby achieving a great simplicity in the interpretation of results (Figure 3.2).

3.4.2.2 Performance Measures and Test Scenarios

The main objective of simulations has been analyzing and comparing the performance of both distributed consensus algorithms in a quantitative manner. For this purpose, we have defined two metrics:

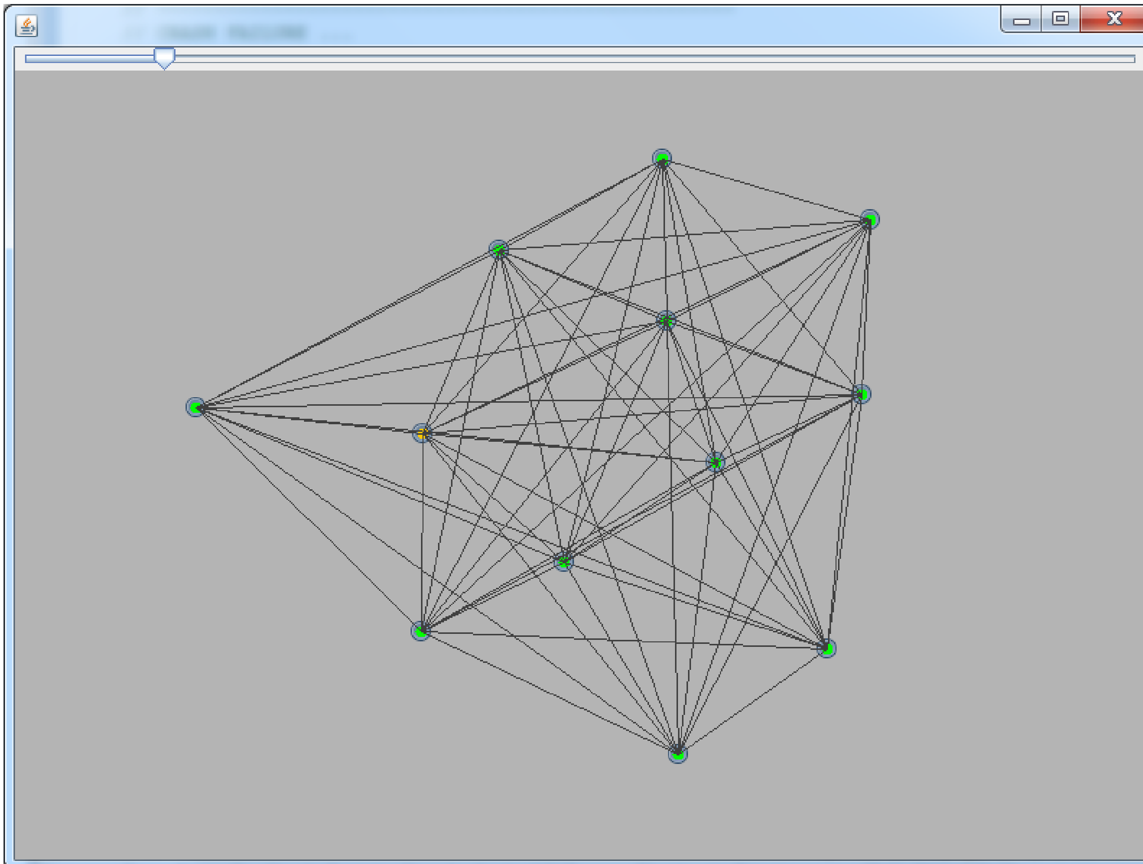


Figure 3.2: The JBotSim Library

- *Early Latency.* We define *latency* as the time elapsed between the beginning of the algorithm and the instant at which the first process decides. This definition of latency, also known as *early latency*, is interesting as a performance metric from the point of view of applications relying on TrustedPals, e.g., fault-tolerant distributed services implemented by active replication, which usually wait until the first reply is returned.

- *Message Complexity.* The second metric, *complexity*, is also known as *message complexity*; it is defined as the number of messages exchanged during the execution of the consensus algorithm. *Message complexity* is of interest for the deployment of TrustedPals in a concrete system.

On the other hand, we contemplate the following four scenarios in the conducted simulations, each one depending on the type of failure that suffers the process that coordinates or leads one round in the consensus:

- *Failure-free.* It is a scenario without failures in the system processes.

- *Crash.* A scenario where the process that coordinates or leads the round suffers a crash failure and never recovers.

- *Omission.* A scenario where the process that coordinates or leads the round suffers omissions of messages (send/receive).
- *Crash-recovery.* A scenario where the process that coordinates or leads the round, suffers a crash failure and after a period of (unknown) time, recovers.

It should be noted that we measured the latency and complexity after the system reaches its steady-state, i.e., the failure detector module into each process permanently suspects all faulty processes, and no correct process is wrongly suspected. In the case of Ω_{Om}, that every failure detector module has chosen as leader the same correct process, i.e., *all correct processes trust the same correct process as the leader.*

3.5 Experimental Results

We analyzed the net effects on the improvement and the overhead of the average latency through the practical comparison in several types of failues. Note that we will refer to Chandra-Toueg consensus algorithm, as Chandra-Toueg and/or CT in tables and figures. For its part, the Paxos consensus algorithm hereafter will be referred to as Paxos.

We have conducted simulations consisting of several sequences of 100 consensus executions for different numbers of processes and scenarios of failure. Another important observation for this study is that the overhead of the network has not been considered.

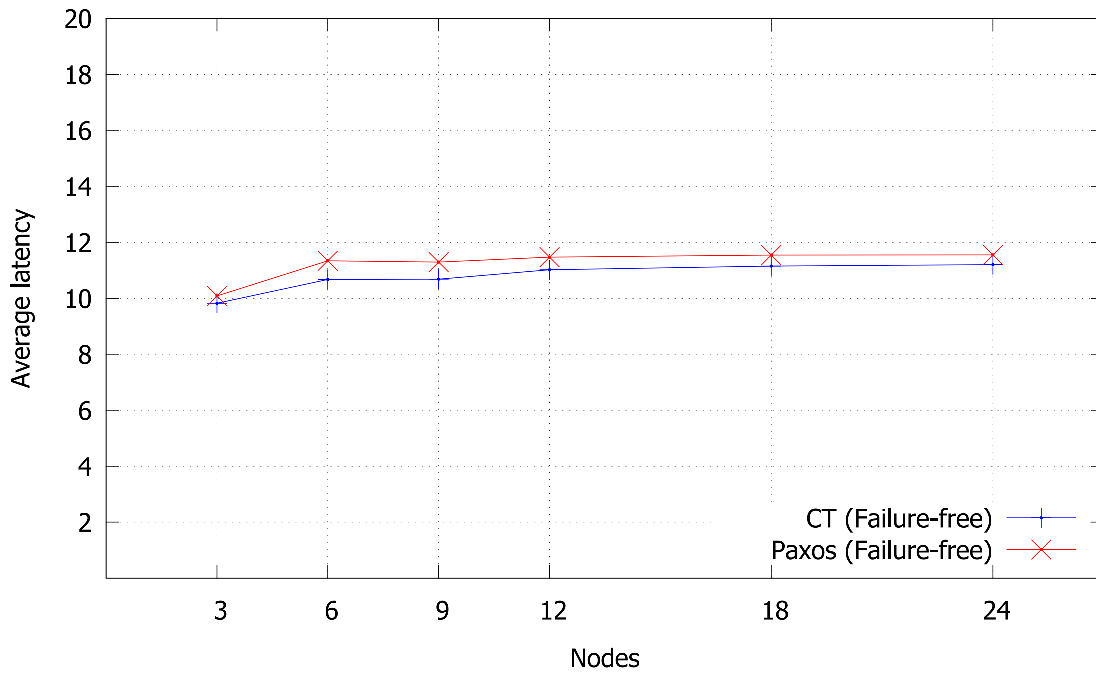
3.5.1 Impact of Scalability on Average Latency

Figures 3.3a and 3.3b show the scalability analysis of the average latency for a number of processes ranging from 3 to 24. Observe that the algorithms present good scalability, both for the failure-free and crash scenarios. It is worth mention that the omission and crash-recovery scenarios not shown (in the previous figures) have similar scalability. Therefore, we consider adequate the number of 12 processes for the rest of the simulations and results that are exposed in this section.

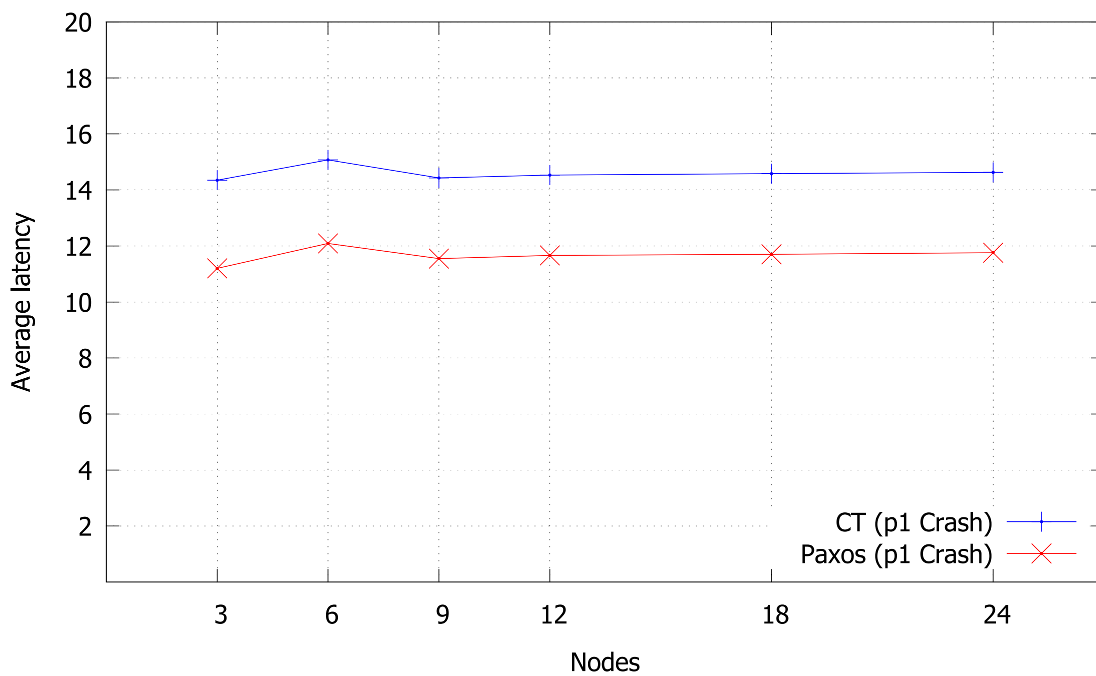
3.5.2 Understanding the Behavior of Average Latency

Table 3.1 and Figures 3.4a, 3.4b, 3.4c, and 3.4d present the behavior of the average latency for the consensus algorithms described above (CT and Paxos).

Table 3.1 shows the average latency and improvement of Paxos versus Chandra-Toueg. We can see that Paxos has a better performance for the crash (19.75%), omission (29.19%) and crash-recovery (29.13%) scenarios, and a slightly worse performance when there is no failure (-4.08%). Observe that the average latency of Paxos is almost steady in a range of 12 ms simulation



(a) Failure-free scenario



(b) Crash scenario

Figure 3.3: Impact of Scalability on Average Latency

Table 3.1: Improvement of Paxos+Ω_{Om} vs CT+$\Diamond\mathcal{P}_{om}$

Scenario	CT+$\Diamond\mathcal{P}_{om}$	Paxos+Ω_{Om}	Improvement
Failure-free	11.02	11.47	-4.08%
Crash	14.53	11.66	19.75%
Omission	16.51	11.69	29.19%
Crash-recovery	16.58	11.75	29.13%

time units approximately in the four scenarios. On the other side, the behavior of the average latency for Chandra-Toueg grows from 11.02 ms in the failure-free scenario to 16.58 ms in the crash-recovery scenario. It is due to the fact that Paxos handles more efficiently the leader election mechanism, unlike Chandra-Toueg wherein each new execution of the algorithm, the failed process is tried as the coordinator of the first round.

Furthermore, we can see in Figures 3.4c and 3.4d that the omission and crash-recovery scenarios show similar behavior for both algorithms. It is because in the $\Diamond\mathcal{P}_{om}$ failure detector, as well as in Ω_{Om}, processes suffering omissions are discarded to be the coordinator (or leader). Moreover, a crashed process that later recovers can be considered as a process that suffers omissions so that it will be treated as an omissive process by the failure detectors ($\Diamond\mathcal{P}_{om}$ and Ω_{Om}). As a consequence, both scenarios present a similar average latency.

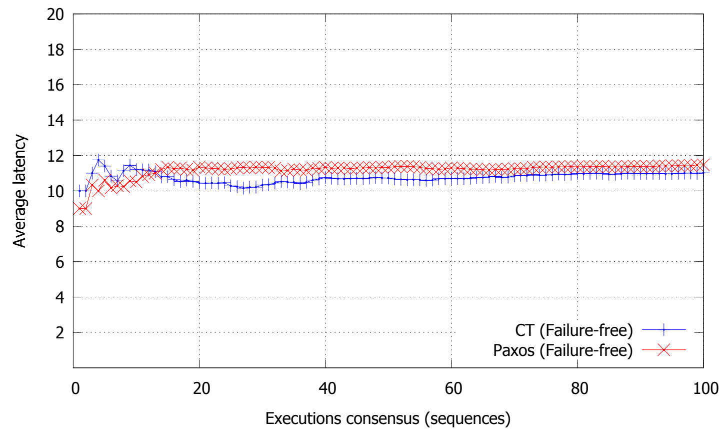
Additionally, we analyzed the overhead of each algorithm regarding the failure-free scenario. The results are presented in Table 3.2 and Table 3.3. Observe that the results from the current architecture (Chandra-Toueg and $\Diamond\mathcal{P}_{om}$) show a performance overhead of 31.85%, 49.82%, and 50.45% for the crash, omission, and crash-recovery scenarios compared to the failure-free scenario. On the other hand, results from the architecture based on Paxos exhibit a slight performance degradation compared with the failure-free case. The reason is that Chandra-Toueg relies on the rotating coordinator paradigm, while Paxos relies on a (more efficient upon failures) eventual leader election mechanism.

Table 3.2: Overhead of CT+$\Diamond\mathcal{P}_{om}$

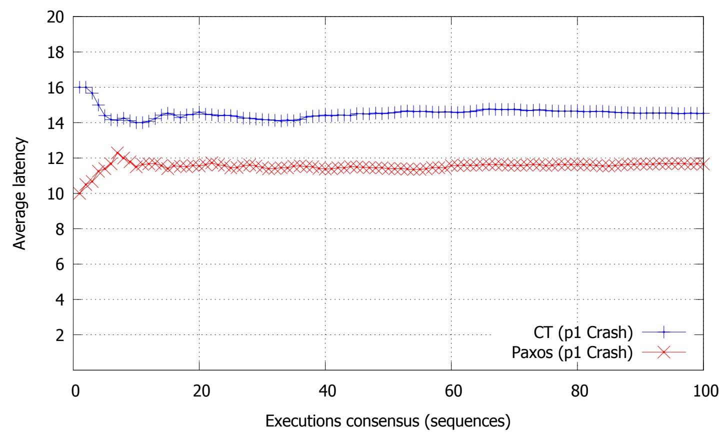
Scenario	Latency	Overhead
Failure-free	11.02	-
Crash	14.53	31.85%
Omission	16.51	49.82%
Crash-recovery	16.58	50.45%

Table 3.3: Overhead of Paxos+Ω_{Om}

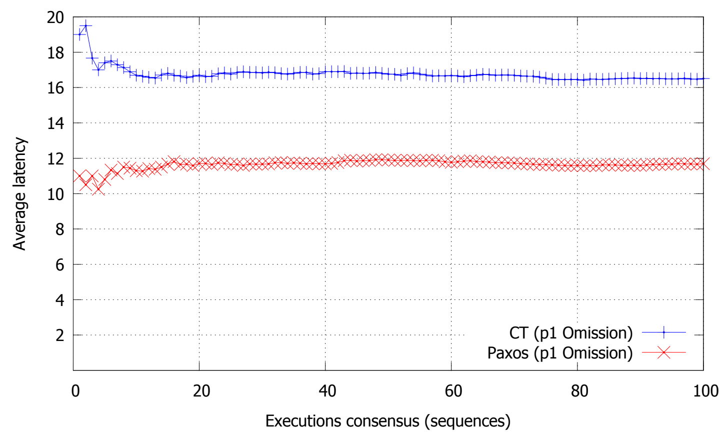
Scenario	Latency	Overhead
Failure-free	11.47	-
Crash	11.66	1.66%
Omission	11.69	1.92%
Crash-recovery	11.75	2.44%



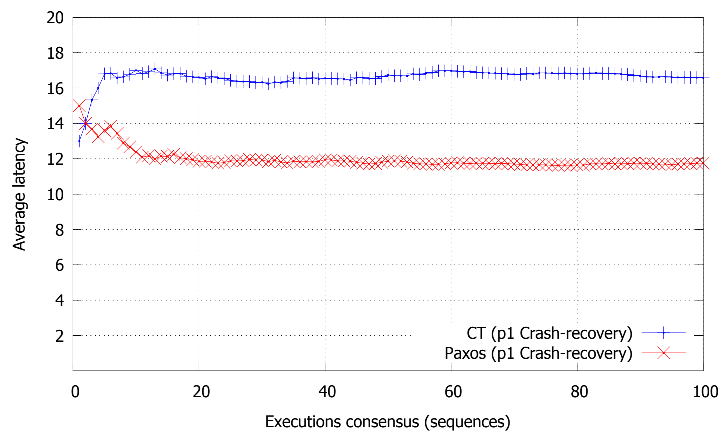
(a) Failure-free scenario.



(b) Crash scenario.



(c) Omission scenario.



(d) Crash-recovery scenario.

Figure 3.4: Average Latency in Different Scenarios

With respect to the complexity, Table 3.4 shows the complexity and improvement of Paxos versus Chandra-Toueg. Once more, we can see that Paxos has better performance for the crash (23.5% of improvement), omission (36.5% of improvement) and crash-recovery (36.5.5% of improvement) scenarios. However, it presents a slightly worse performance when there is no failure (-14.9%). On the other hand, the current architecture (Chandra-Toueg and $\diamond\mathcal{P}_{om}$) shows a performance overhead of 45%, 81% and 81% for the crash, omission and crash-recovery scenarios compared to the failure-free scenario, quite the contrary with the presented in the proposed architecture based on Paxos+Ω_{Om}, with similar cost in the four scenarios.

Table 3.4: Complexity Improvement of Paxos+Ω_{Om} vs CT+$\diamond\mathcal{P}_{om}$

Scenario	CT+$\diamond\mathcal{P}_{om}$	Paxos+Ω_{Om}	Improvement
Failure-free	47	55	-14.9%
Crash	67 (+45%)	52 (-3%)	23.5%
Omission	85 (+81%)	54 (0%)	36.5%
Crash-recovery	85 (+81%)	54 (0%)	36.5%

3.5.3 System under Multiple Simultaneous Failures

We have performed a comparison for a different number of processes suffering failures simultaneously. Figure 3.5 shows the behavior of the average latency in the crash scenario for 1, 2, and 3 failures, respectively.

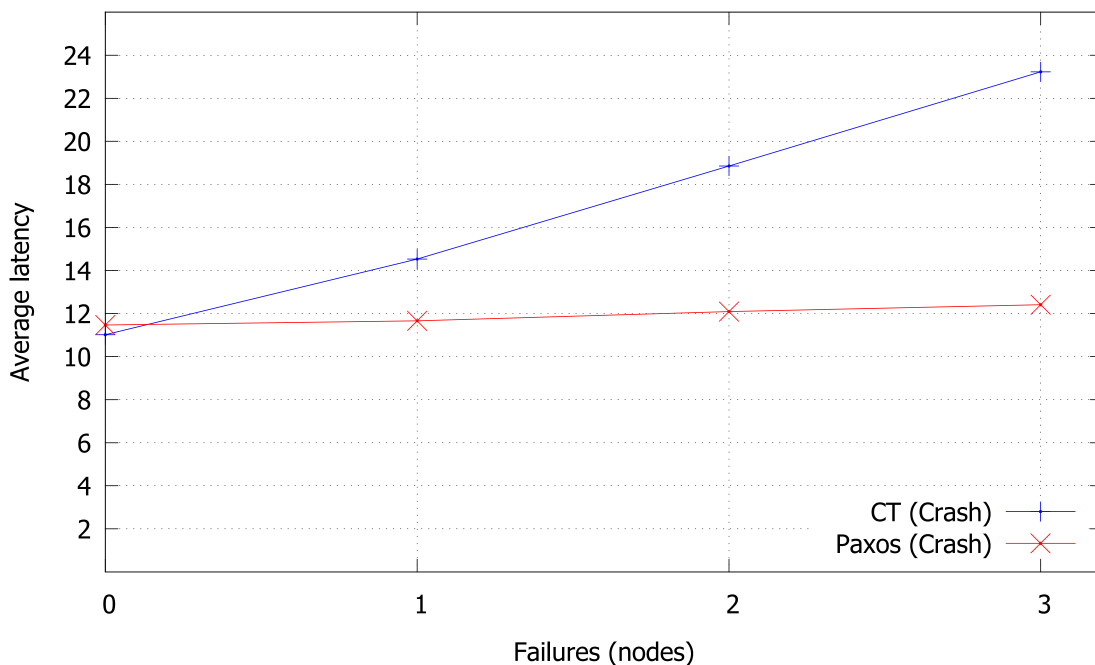


Figure 3.5: Comparison for Different Numbers of Failures (*Crash scenario*)

Table 3.5 presents the overhead in each algorithm regarding the failure-free scenario. Besides, we can compare the overhead suffered between algorithms. The effect of several process failures

Table 3.5: Overhead of Paxos+Ω_{Om} vs CT+$\diamond\mathcal{P}_{om}$ (*multiples failures*)

Scenario	CT+$\diamond\mathcal{P}_{om}$	Paxos+Ω_{Om}
p_1 crash	31.85%	1.66%
p_1 and p_2 crash	71.05%	5.41%
p_1, p_2 and p_3 crash	110.80%	8.20%

is very harmful to Chandra-Toueg and directly affects its average latency (overhead of 110.80% when the first three processes crash), due to the increased number of rounds required to reach consensus, i.e., when the first three processes have failed (p_1, p_2, and p_3 crash) it is necessary to execute at least four rounds.

On the other hand, when a new consensus with a stable leader starts in Paxos, then it always succeeds in the first round, and hence the overhead of Paxos will not be substantially affected (overhead of 8.20% when the first three processes crash).

To finish, we reinforce the idea: *the architecture based on Paxos exhibits slight performance degradation compared with the architecture based on Chandra-Toueg.* The reason is that the eventual leader mechanism that uses Paxos is more efficient on different scenarios of failures, in comparison with the rotating coordinator mechanism used in Chandra-Toueg (see Figure 3.6).

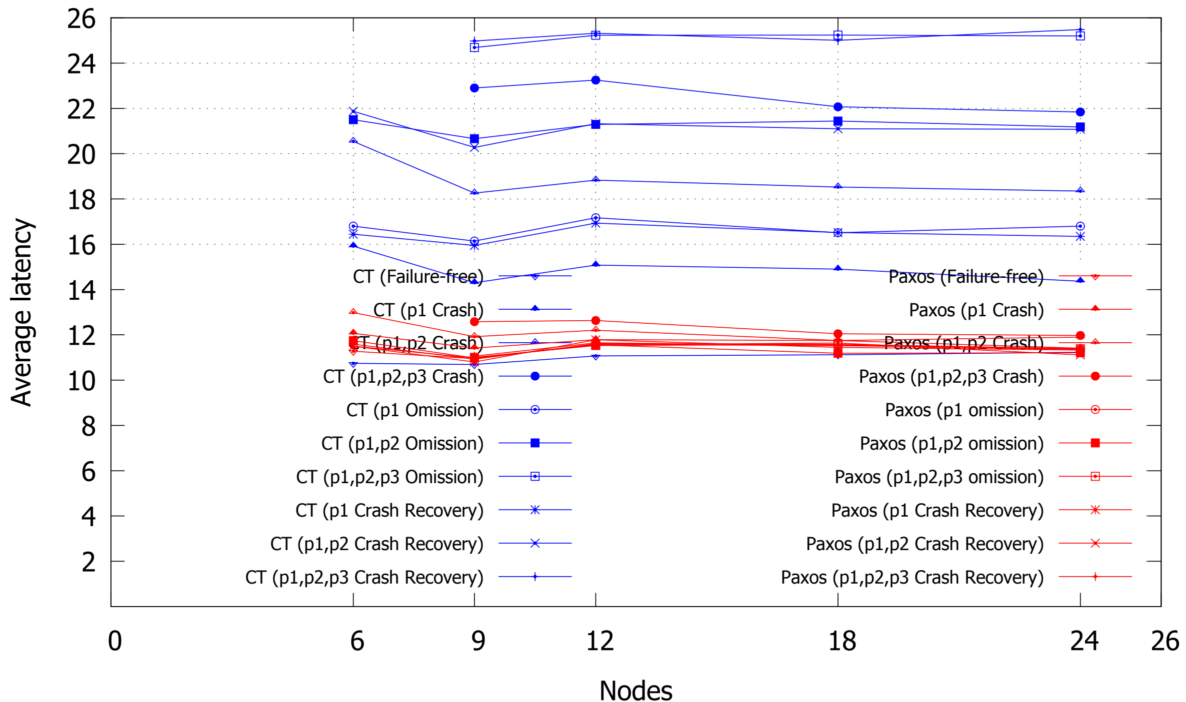


Figure 3.6: Different Failures Scenarios

3.6 Chapter Summary

In this chapter, we have analyzed two well-known distributed consensus algorithms: Chandra-Toueg and Paxos. By using the TrustedPals framework as a case of study, we presented a modular manner of building distributed consensus algorithms in the partial synchrony model.

We have realized a practical performance study of these two indulgent consensus algorithms [Gue00, GR04] upon crash-recovery and message omission failures. Both algorithms execute in rounds, and the main difference among them is the way each round is coordinated. Chandra-Toueg is based on the rotating coordinator paradigm, while Paxos is based on a leader election mechanism.

As experimental results, we showed the performance of both algorithms under different scenarios of failures. Given the results obtained, we believe that Paxos exhibits significant advantages compared to Chandra-Toueg in dynamic and highly fault-tolerance scenarios.

Therefore, we can summarise:

- We have presented a performance study through computational simulation tools of these two consensus algorithms under different failure scenarios. Note that both (indulgent) algorithms are augmented with an unreliable failure detector ($\diamond\mathcal{P}_{om}$ and Ω_{Om}, for Chandra-Toueg and Paxos, respectively).
- We have identified and measured two key performance metrics for this study, *early latency* which reflects the time that the algorithm needs to reach a common decision, and on the other hand, *message complexity* that is the number of messages required to reach consensus. We conducted several executions of both algorithms in different scenarios, and we analyzed the average early latency and average message complexity.
- Our results show that Paxos is more efficient than Chandra-Toueg if the first process that coordinates or leads one round suffers a failure (crash, omission, or crash-recovery), given that Paxos handles the eventual leader election mechanism efficiently. Also, we can indicate that the results corresponding to the failure-free and crash scenarios are qualitatively similar to those of the work by Hayashibara et al. [HUSK02] and Urban et al. [UHSK04] (in those works, the omission and crash-recovery scenarios were not considered).

4 Distributed Eventual Leader Election Service

"We must use time wisely and forever realize that the time is always ripe to do right."
— Nelson Rolihlahla Mandela[4]

An eventual leader service is essential to design and implement reliable applications on top of an asynchronous distributed system prone to failures. A manner to build an eventual leader service is by solving the leader election problem. It allows solving several fault-tolerant agreement problems such as *the consensus problem.* The purpose of the leader election is to choose a single correct process that will coordinate the actions of the distributed system. The interest in studying this problem lies in the difficulty of resolving it in an asynchronous environment, addressing different types of failures.

Outline. The Chapter is organized as follows. Leader election in distributed systems is presented in Section 4.1. Section 4.2 describes the system model and considered assumptions. The eventual leader election problem is presented in Section 4.3; in particular, the specification of an eventual leader election service. Section 4.4 describes the pitfalls of the system to be considered. In Section 4.5, three novel approaches are proposed which implement a distributed eventual leader service (*Basic 4.5.1, Communication-Efficient 4.5.2, and Indirect-Leader Trusting Mechanism 4.5.3*). Finally, Section 4.6 summarizes the chapter.

4.1 Leader Election in Distributed Systems

In asynchronous distributed systems, it is complicated to evaluate if one of the processes involved in a calculation operates correctly or has failed. Unreliable failure detectors were proposed to deal with this problem. In essence, an unreliable failure detector [CT96] in an abstract module which encapsulates the timing assumptions necessary to assess the operating status of a process. A particular type of failure detector is Omega [CHT96], which outputs a (common) single process that has not failed, allowing the implementation of a leader election functionality.

[4] South African president, leading figure in anti-apartheid movement (1918–2013).

More than simply marking failed processes, unreliable failure detectors capture in an abstract way timing assumptions necessary to the correct operation of many distributed algorithms [CT96, Lam98]. The lack of reliability of these detectors is key to this abstraction, i.e., mistakes can be made, and failures are detected eventually in a way that it reflects the intrinsic time uncertainty in asynchronous distributed systems.

Leader election is a fundamental problem in distributed systems since it allows processes in the system to select a unique process (*leader*) among them. The leader will be able to coordinate distributed actions, allowing to solve several fault-tolerant agreement problems such as consensus [PSL80, BM93].

4.1.1 Synchrony and Failure Models

Regarding the assumptions of time and its constraints, it can be described as follows [Ray10, CGR11].

Asynchronous distributed systems have no bounds with regard to *(i) message delay*, how much time it takes for a message to be delivered, and *(ii) processing time*, how much time it takes for a process to do some computation.

Synchronous distributed systems can rely on hard bounds for message delay and computing time.

In partially synchronous distributed systems [CT96], we consider processes and links behave most of the time asynchronously, but there is an unknown time after which they behave synchronously.

Another important characteristic that defines a distributed system is the type of failures that the processes in the system can suffer.

4.1.2 Resilience to both Crash-Recovery and Omissions Failures

In the crash failure model [LSP82, SS83, HT93, CGR11], a correct process is supposed never to crash. Once crashed, a process is deemed to be faulty, and it never recovers. On the other hand, the crash-recovery failure model considers correct a process that never crashes, or crashes and recovers a finite number of times. Thus, in the crash-recovery failure model [HMR98, ACT00, MLJ09, CGR11], a faulty process is a process that crashes and never recovers or a process that crashes and recovers infinitely often.

We have a particular interest in designing and implementing a leader election service that addresses both crash-recovery and omission failures. Observe that, under some specific conditions, e.g., assuming that the state of a process is never lost, each failure model could be treated as a particular case of the other. An omission failure could be modeled as a crash before the omission followed by a recovery, and a crash-recovery failure could be modeled as an omissive period. Nevertheless, *the systematic use of stable storage can be very restrictive*, regarding both space and latency [CGR11].

4.1.3 About the Detectability of Failures

We concentrate on providing a leader election service based on the detectability of failures at each process (i.e., completeness and accuracy). Therefore, when we consider different failures in a process, there are some interesting remarks to keep in mind.

- *There are types of failures that can not be detected.* For example, if a process p suffers send-omissions towards a crashed process, then these omissions will be *undetectable*, and p could be considered a correct process.
- *Some failures can be detected, but the type cannot be deterministically identified.* Note that sometimes, there is no way to identify which type of failure a faulty process has suffered. If a process p does not receive any more expected messages from another process q, it could be due to the fact that q is suffering permanent send-omissions towards p. Nevertheless, it is also possible for q to have Byzantine behavior. Both cases could be *indistinguishable.*
- *Instead of correct processes, sometimes it is interesting to detect good processes.* In the crash model, faulty processes stop computing and communicating. However, in other models as crash-recovery and omission, a faulty process could still be operational as long as it keeps a minimal ability to compute and communicate in order to solve a specific problem. The concept of good process underlies in the work of Guerraoui et al. [GHM+99].

4.1.4 Failure Detection and Leader Election

A distributed system is composed of processes and communication links. A failure detector is an abstract component that outputs not necessarily correct information about which processes are correct or faulty. Failure detectors work locally within each process in the system, and usually, they are implemented by exchanging messages.

A significant result is that it is possible to build reliable distributed systems on top of an unreliable failure detector [CT96]. It means that a failure detector is not supposed to be correct while the system behaves asynchronously [FLP85], since it may make mistakes by suspecting correct processes or trusting faulty ones. Despite the mistakes it can make, a failure detector is a powerful abstraction as it encapsulates unpredictable system behavior.

Of particular interest for us is the Omega failure detector (Ω), initially presented in [CHT96]. Chandra et al. [CHT96] showed that Ω is the weakest failure detector for solving consensus in partially synchronous systems with processes that can suffer failures, encapsulating the synchrony assumptions to circumvent the *FLP impossibility result* [FLP85].

The Omega failure detector is considered as a leader election mechanism, a failure detector that outputs a single trusted process. Formally, the Ω failure detector is specified by the following properties [Gue00]:

(i) *Eventual Accuracy.* There is a time after which every correct process trusts some correct process.

(ii) *Eventual Agreement.* There is a time after which no two correct processes trust different processes.

Both properties guarantee that every correct process will eventually trust the same correct process. The eventual behavior means that it is necessary a long enough period of synchrony for the properties to be accomplished.

The Ω failure detector is used as a building block to solve more complex problems such as consensus [Lam98] and atomic broadcast [CHT96]. The Ω failure detector was devised for a crash failure model, in which any number of processes can fail by prematurely by halting (and not recovering). Afterward, Ω has been widely studied as a leader election, e.g., [Lam98, LFA00, MR01, MOZ05, MRT06, ADFT08, AJR10, LRAC12].

Later works extended the Ω failure detector study to the crash-recovery failure model. In addition to the crash failure model, in the crash-recovery failure model, crashed processes may recover and still participate. In this failure model, correct processes are often redefined as those processes that crash and recover a finite number of times, i.e., processes that eventually do not crash again. Observe that this failure model subsumes the crash failure model. Martin et al. [MLJ09, ML10] and Larrea et al. [LMA11] have proposed several Ω algorithms for the crash-recovery failure model.

We also consider the general omission failure model, in which processes may suffer omissions while sending or receiving messages. Several failure detectors for this failure model have been proposed as well [SCL+11, CFGA+12]. More precisely, Delporte-Gallet et al. [DGFF05] proposed an Ω for the crash and general omission failure model. Lately, Fernandez-Campusano et al. [FCL14a, FLCR15, FLCR16, FLCR17] proposed a novel definition of Ω allowing some "*incorrect*" processes to participate in the distributed agreement of a common leader and hence, to participate actively in the consensus.

4.1.5 Communication Efficiency

Furthermore, we also look for an efficient leader election service regarding communication efficiency. In this sense, we define a communication-efficient implementation of leader election if and only if: *(i)* At most, $(n - 1)$ communication links are used permanently, and *(ii)* from some point on, only one process (*the leader*) keeps sending messages [ML10, LMA11].

4.2 System Model and Assumptions

We consider a message-passing system composed of a finite and totally ordered set of n processes denoted by Π. Processes only communicate by sending and receiving messages. Every process

is connected with every other process by communication links. Communication links are reliable and cannot create or alter messages.

Regarding the timing assumptions. Like our performance study of consensus algorithms (see Chapter 3.3), as timing assumptions, we assume a partially synchronous system based on the model of Chandra and Toueg [CT96], i.e., there exist upper bounds on processing time and message communication delay, although those bounds are not known a priori by processes. So, the *partial synchrony model* that we consider can be summarized as:

- We define as: Time $\implies$ Δ (message delay) + ϕ (processing time)
  - Δ, upper bound on message delay (*unknown*)
  - ϕ, upper bound on processing time (*unknown*)
- We assume that: exists a Global Stabilization Time (GST), such as:
  - Before GST, the system presents an *asynchronous* behavior, i.e., bounds do not hold ($\Delta + \phi$)
  - After GST, the system presents a *synchronous* behavior, i.e., bounds hold ($\Delta + \phi$)
  - GST is *unknown*

Regarding the failure model. Processes can fail by crashing and by omitting messages. Crashes are not permanent, i.e., crashed processes can recover and participate again in the system.

Regarding the process classes. In every execution of the system, we have the following three types of processes [MLJ09]:

(1) *eventually up*, i.e., processes that eventually remain up forever (including processes that never crash, also named *always up*),

(2) *eventually down*, i.e., processes that eventually remain crashed forever (including processes that never start, also named *always down*), and

(3) *unstable*, i.e., processes that crash and recover an infinite number of times.

In addition to crash-recovery failures, processes can also fail by omission at sending and/or receiving messages [PT86]. Omissions can be selective, i.e., concerning some given or all processes. Also, omissions can be transient or permanent.

Indulgent consensus algorithms [Gue00, GR04] require a *majority of correct processes.* Hence, we consider that correct processes are those *eventually up* processes that eventually communicate without omissions with *a majority of eventually up processes* in the system.

Remark. The algorithms which are detailed below (see Section 4.5) have some differences regarding the general system model, and the differences reside in:

- The communication link between two processes can be either *unidirectional* or *bidirectional.*
- Communication links can be either *reliable* or *eventually reliable*, and cannot create or alter messages.
- The presented algorithm can make *use or not use of stable storage* to store the number of times that the process suffers crash-recovery failures.

4.3 Specifying an Eventual Leader Election Service

The election of an eventual leader in an asynchronous system prone to failures is an important problem of fault-tolerant distributed computing. This problem can be solved by the implementation of the Ω failure detector, which provides a distributed eventual leader election service.

The problem lies in that a service of this type cannot be built if the underlying system is fully asynchronous, due to the impossibility result (*FLP impossibility result* [FLP85]), to solve consensus in a deterministic way when one process suffers a failure.

The leader election problem can be solved through a leader election service with characteristics of eventuality in the time. An *eventual leader election service* [Ray07] may be defined as follows:

(i) Each time it is called, it returns the identity of a process (p_{id}), and

(ii) after some finite time, it always returns the same (p_{id}) corresponding to the identity of a *correct process.*

We want to provide access to an eventual leader election service to each process (that composes a distributed system), with the idea of chasing the weakest system model for implementing Ω as a distributed eventual leader election service. Such that, it allows for improving the design and implementation of applications in asynchronous distributed systems prone to failures.

The interest in studying the problem lies partly in the difficulty of solving in asynchronous environments by addressing different types of failures. In this environment, the processes can fail by crashing, or by crashing and later recovering (several times). In addition to these failures, a process may suffer either a send-omission or receive-omission or both, also permanent omissions or transient omissions, non-selective omissions, or selective omissions.

Two important points to remember (see Section 2.2.5):

- When a crashed process recovers, it may lose important information about what it has learned or done before crashing. To avoid this, sometimes processes are enhanced with additional stable storage where important information can be stored and later accessed when process recovers.

- A process suffers a send-omission failure if it executes a message send operation, but the message never reaches the communication link. A process suffers a receive-omission when a message is received at its destination process, but at this process, the message is never delivered. A permanent omission occurs when a process suffers an omission message so that every subsequent message will be omitted. Moreover, a process that suffers a transient omission can send/receive messages reliably again until another failure occurs.

4.3.1 From Ω to an Eventual Leader Election Functionality

When the Ω failure detector was proposed by Chandra et al. [CHT96], it assumed only the crash failure model. In this case, it focuses on properties that correct processes eventually fulfill (note that, non-correct processes eventually crash). On the other hand, when considering the crash-recovery failure model, correct processes are usually redefined as *eventually up* processes, although other works consider other definitions, e.g., *good* processes [ACT00]. Those processes should eventually output a common leader, while a *null* value (denoted $\perp$) is used to prevent processes without a leader from disagreeing.

In addition to crash-recovery failures, we consider that processes may also suffer message omissions. This way, an *eventually up* process p could suffer omissions in its communication links with the rest processes, which could prevent process p from communicating with the rest of the processes in the system. Such a situation could produce undesirable effects in the system. So, we must provide new definitions and properties of the Omega failure detector as eventual leader election functionality to a new environment of failures.

We refer to an eventual leader election functionality when every process in the system is augmented with a module, such that it returns a process identifier as a leader estimation (a single correct process). Then, Ω must ensure that, after some time τ, the module at each correct process will eventually provide the same leader identifier (a non-faulty process). However, *(1) there is no knowledge about when τ happens* and *(2) before τ there may be more than one leader.*

4.3.1.1 A Novel Definition for Ω

Formally, the Omega failure detector (Ω) has been defined as follows [CHT96]:

Definition 1 (Omega$_{crash}$). *There is a time after which every correct process always trusts the same correct process.*

Observe that the previous definition, made for the crash failure model, does not state anything about incorrect processes, which are allowed to disagree at any time with correct processes. This fact can affect negatively an attempt to solve consensus due to the existence of several leaders (the termination of leader-based consensus relies on the eventual existence of a unique alive leader). However, in the crash failure model, incorrect processes eventually crash, so

there will eventually exist a unique leader. Contrary to this, in the crash-recovery and omission failure models, there can be incorrect processes running forever (e.g., unstable processes).

Clearly, processes that, due to continuous crash-recovery or permanent omissions, become disconnected from correct processes cannot be forced to agree on the correct leader. Hence, we consider the following alternative definition for Ω, proposed for the crash-recovery failure model in [LMA11]:

Definition 2 (Omega$_{crash-recovery,\ omission}$). *There is a time after which (1) every correct process always trusts the same correct process ℓ, and (2) every incorrect process may only alternate between trusting either ℓ or no one ($\perp$, i.e., it does not trust any process).*

This alternative definition of Ω fits better with our system model in which some eventually up processes, as well as unstable processes, may permanently omit messages. Observe also that this definition is very interesting for leader-based consensus algorithms because it allows incorrect processes to delay their participation in the algorithm until they trust a process, thus, ensuring eventual agreement on the leader and making consensus solvable.

4.3.1.2 The Set of Correct Processes

As mentioned, indulgent consensus algorithms require a majority of correct processes. Hence, we assume a majority of processes in the system, named *CORE* (also denoted as *C*), that are eventually up and eventually and permanently do not omit messages among them.

Note that this requires the communication links among processes in the set *CORE* to be eventually reliable. Also, we assume that every eventually up process that eventually and permanently communicates without omission or loss with a majority of processes belongs to the set *CORE*.

Based on the previous, processes in the *CORE* are considered *correct*. The rest of the processes, i.e., eventually up processes not belonging to the set *CORE*, eventually down and unstable processes, are considered *incorrect*, and their links can be lossy.

To formalize the definition of the *CORE* set, first, we define the *connected* relationship:

Definition 3. *Two processes p and q are connected iff p communicates without omissions with q in both directions (from p to q and from q to p).*

Then, we define the *CORE* set as follows:

Definition 4. *We define C as the set of eventually up processes that eventually and permanently are* connected *with a majority of eventually up processes.*

Concerning timeliness, we assume that the set *CORE* is partially synchronous, i.e., there is a time after which there exist upper bounds on processing time and message communication delay, although those bounds are not known a priori by processes. Processes not in *CORE* can behave asynchronously.

4.3.1.3 Eventual Leader Election Properties

Defined the *connected* property (see Definition 3), we can provide the properties of an eventual leader service. Observe that, if and only if p is *connected* then:

(i) p is an *eventually up* process, and

(ii) p does not suffer omissions with a majority of eventually up processes in the system.

It implies that p can suffer a finite number of crash-recoveries and infinite omissions, as long as it remains *connected*. On the other hand, *non-connected* processes include crashed processes, unstable processes and, eventually up processes without enough connectivity.

Based on Definition 4, we present the following two properties for an **eventual leader election service**.

Property 1. $\exists \ell \in C, \forall p \in C$: *eventually and permanently* $leader_p = \ell$.

There exists an eventually up process ℓ such that every *eventually up* process eventually and permanently connected with a majority of processes eventually and permanently considers ℓ as the leader.

Property 2. $\forall q \notin C$: *eventually and permanently: every unstable process* ***only alternates*** *between either* $leader_q = \ell$ *or* $leader_q = \perp$.

There is a time after which, every non-crashed process q outputs as leader only either ℓ or $\perp$.

4.4 Difficulties Underlying System Model Assumptions

Now we describe the pitfalls of the system defined in Section 4.2, and how our proposal copes with them. An eventual leader election service in distributed systems has to manage three main difficulties: *eventual synchrony*, *unstable processes due to crash-recovery failures*, and *communication failures due to selective omissions.*

(A) *Eventual synchrony.* One of the main advantages of the failure detector abstraction is that it encapsulates the timing assumptions of the system. This way, our eventual leader election service allows eventually to select a unique leader in partially synchronous systems. For its implementation, we follow the traditional approach of periodical exchange of *ALIVE* (heartbeat) messages, combined with an adaptive time-out managing and the possibility of retracting false suspicions.

(B) *Unstable processes.* Processes that crash and recover an infinite number of times should not be considered candidates when selecting a leader since we cannot rely on them to reach consensus. To eventually avoid the potentially harmful effect of this kind of processes, we propose to track the number of times each process recovers. This value will eventually stop changing for eventually up processes, and it will be incremented forever in the case of unstable processes, so we propose to consider this value when selecting a leader.

(C) *Selective omissions.* We have differentiated two negative effects of selective omissions:

   (i) Selective omissions may affect the stable leader election if they are not appropriately managed. If a process p with a high potential to be considered leader is getting periodically *connected* and later disconnected from a majority of processes, it should not be considered as the leader, since otherwise, it would prevent the leader election service from eventually providing a stable output. In order to avoid considering as leader processes with such an unstable behavior in terms of communication, we propose to monitor the number of times each process gets *disconnected* from a majority of processes and to consider this measure when selecting a leader.

   (ii) Also, selective omissions may create scenarios where eventually up processes *connected* with a majority of processes do not communicate without omissions with the leader. Figure 4.1 provides an example of such a scenario. There are three processes in the system: p, q and r. p and q are *connected* and p and r too, i.e., all of them communicate without omissions with a majority of processes. When selecting a leader, we use the rank leadership presented previously to avoid unstable behaviors of processes. Numeric values at each process indicate a leadership rank value (a numerical criterion to select the leader, being the lowest value the best rank). According to these values, q should be selected as the leader. Since q and r are not *connected*, we allow processes to communicate (*propagate*) its leader to other processes. This way, after p selects q as the leader, p informs r about the leadership rank of q so that r can also select q as its leader. Since every *connected* process communicates without omissions with a majority of processes, only one level of indirection is needed to reach all connected processes.

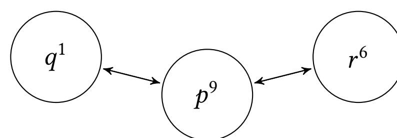


Figure 4.1: Example of three *connected* processes

4.5 Implementing an Eventual Leader Service

In this section, we present three distributed eventual leader election algorithms satisfying Definition 2. These three approaches implement an eventual leader election service (*Basic 4.5.1,*

Communication-Efficient 4.5.2, and Indirect-Leader Trusting Mechanism 4.5.3) for crash-recovery and omission environments to support fault-tolerant agreement algorithms, e.g., the Paxos consensus algorithm.

As a novelty concerning previous works in the literature, our algorithms tolerate the occurrence of both crash-recoveries and message omissions to any process during some finite but unknown time, assuming that eventually, a majority of processes in the system remain up forever and stops omitting messages among them.

Based on our new definition of Omega, we have boosted a new eventual leader election service allowing some "*incorrect*" processes to participate in the distributed agreement of a common leader and hence, take an active part in the consensus.

4.5.1 Basic Eventual Leader Election

Our first proposal consists of a basic distributed leader election algorithm for crash-recovery and omission environments. The current algorithm satisfies the definitions and properties presented in Section 4.3.1.

It is worth mentioning that the considered system model is the one presented in Section 4.2, although we must stipulate the following complementary assumptions:

- The algorithm *assumes the availability of stable storage* to store the number of times that the process suffers crash-recovery failures.
- Every pair of processes is *connected* by two unidirectional *eventually reliable communication links*, one in each direction.

4.5.1.1 Scenario of System

Algorithms 4.1, 4.2 and 4.3 show the proposal of a *Basic Eventual Leader Election* in detail.

Roughly speaking, as shown in Figure 4.2, every process p keeps track of its communication with every other process by exchanging *ALIVE* messages periodically. In case a loss/omission is detected, communication in that link is marked as suspicious. In order to mend transient omissions, *lost* message(s) are requested to be sent again. So, to select a leader, every process first checks whether it communicates well with a majority of processes. In that case, it selects the process q in that majority that has the smallest *penalty* value. If that process considers itself as the leader, then p sets q as its leader. Otherwise, no leader is set ($\perp$ value).

4.5.1.2 The Algorithm

Basic Eventual Leader Service is composed of an initialization part (Algorithm 4.1), and four concurrent tasks (Algorithm 4.2) and a procedure for determining leader (Algorithm 4.3), which we describe in the following paragraphs.

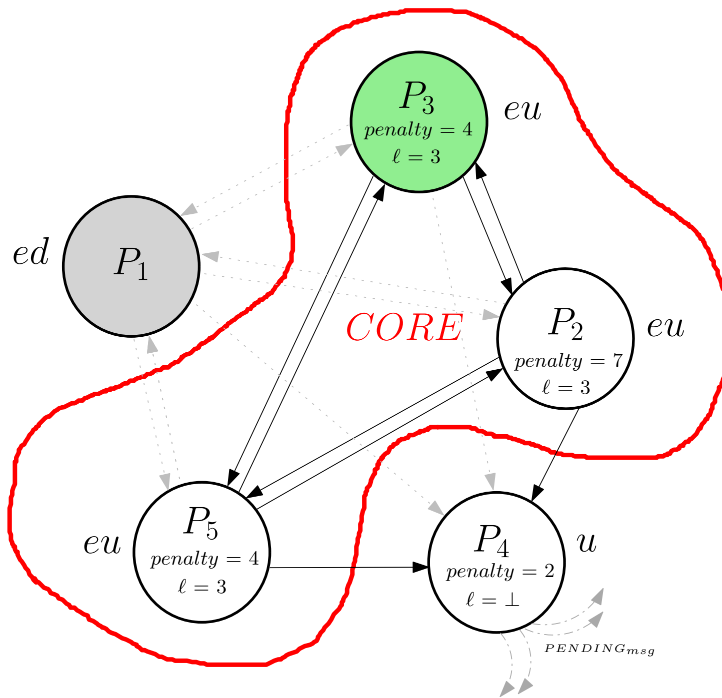


Figure 4.2: Basic Eventual Leader Election *(Scenario of System)*

1 || **Initialization:** *{Executed by process p upon start/recovery}*
2 **if** *first execution of the algorithm* **then** $penalty_p[p] \leftarrow 0$
3 $penalty_p[p] \leftarrow penalty_p[p] + 1$ *{$penalty_p[p]$ is in local stable storage}*
4 $idMsgSent_p \leftarrow 0$
5 **for** *all* $q \in \Pi$ *except* p **do**
6     $penalty_p[q] \leftarrow 0$
7     $idMsgReceived_p[q] \leftarrow 0$
8     $wellConnectedIn_p[q] \leftarrow FALSE$
9     $wellConnectedOut_p[q] \leftarrow FALSE$
10     $leaderCandidates_p[q] \leftarrow FALSE$
11 $wellConnectedIn_p[p] \leftarrow TRUE$
12 $wellConnectedOut_p[p] \leftarrow TRUE$
13 $leaderCandidates_p[p] \leftarrow TRUE$
14 $\ell_p \leftarrow \perp$
15 **for** *all* $q \in \Pi$ *except* p **do**
16     $Timeout_p[q] \leftarrow \eta$
17   reset $timer_p[q]$ to $Timeout_p[q]$
18 **start** tasks ***Task 1, Task 2, Task 3*** **and** ***Task 4***

Algorithm 4.1: Basic algorithm: process initialization.

```
19 || Task 1: Repeat forever every η time units          {periodical sending of ALIVE messages}
20    idMsgSent_p ← idMsgSent_p + 1
21    for all q ∈ Π except p do
22       send(ALIVE, p, penalty_p[p], idMsgSent_p, wellConnectedIn_p[q], ℓ_p) to q

23 || Task 2: When receive (ALIVE, q, penalty_q, idMsgSent_q, wellConnected_q, ℓ_q)      {ALIVE }
24    if penalty_q > penalty_p[q] then                   {if q has restarted since last message}
25       penalty_p[q] ← penalty_q
26       idMsgReceived_p[q] ← idMsgSent_q − 1
27    if penalty_q = penalty_p[q] then
28       if idMsgSent_q = idMsgReceived_p[q] + 1 then    {if expected message}
29          reset timer_p[q] to Timeout_p[q]
30          idMsgReceived_p[q] ← idMsgSent_q
31          wellConnectedIn_p[q] ← TRUE                  {communication from that process is OK}
32          wellConnectedOut_p[q] ← wellConnected_q      {learn from q if it is well connected}
33          leaderCandidates_p[q] ← (q = ℓ_q)            {learn from q if it considers itself as a leader}
34          UpdateLeader()                               {process changes in leadership}
35       else
36          send(PENDING, p, penalty_p[q], idMsgReceived_p[q]) to q      {Ask for misplaced msgs}

37 || Task 3: When receive (PENDING, q, penalty_q, idMsgReceived_q)      {misplaced msgs request}
38    if penalty_q = penalty_p[p] then
39       for all msgid from idMsgReceived_q + 1 to idMsgSent_p do
40          send(ALIVE, p, penalty_p[p], msgid, wellConnectedIn_p[q], ℓ_p) to q

41 || Task 4: Upon expiration of timer_p[q]              {ALIVE message not received in time}
42    Timeout_p[q] ← Timeout_p[q] + 1
43    wellConnectedIn_p[q] ← FALSE
44    leaderCandidates_p[q] ← FALSE
45    UpdateLeader()
```

Algorithm 4.2: Basic algorithm: main tasks.

Initialization. During initialization, Lines 1 to 18, every process p first checks whether it is the first execution of the algorithm, in which case it initializes its penalty value to 0 (Line 2). Then, p increments its penalty value by 1 (Line 3). It is important to note that, although penalty values of all processes in the system will be locally stored by p in a vector called $penalty_p$, only p's own penalty, i.e., $penalty_p[p]$, is stored in stable storage. In order to detect message losses/omissions, processes will send messages with an increasing sequence number. This sequence number, $idMsgSent_p$, is initialized to 0 (Line 4). Next, the $penalty_p$ vector is initialized to 0 for the rest of the processes q (Line 6). The vector that is used by p to store the sequence number of the last message received from every other process q, named $idMsgReceived_p$, is initialized to 0 (Line 7). Two Boolean vectors, named $wellConnectedIn_p$ and $wellConnectedOut_p$, are used to indicate if p is well connected at reception and at sending with every other process, respectively. They are initialized to *FALSE* (Lines 8-9). Finally, another vector $leaderCandidates_p$, initialized to *FALSE* (Line 10), is used to indicate if a process considers itself as a candidate to

```
46 || Procedure UpdateLeader:            {select a leader based on collected information}
                                          {First, calculate the set of wellConnected processes}
47    C_p ← ∅
48    for all q ∈ Π do
49      if ( wellConnectedIn_p[q] = TRUE and
             wellConnectedOut_p[q] = TRUE ) then insert q into C_p
              {Now, select the leader among well connected processes which considers themselves as leader}
50    ℓ_p ← ⊥
51    if |C_p| > n/2 then
52      q ← select process ∈ C_p with smallest penalty_p[q], using the process id to break ties
53      if leaderCandidates_p[q] = TRUE then ℓ_p ← q
54    else
55      penalty_p[p] ← penalty_p[p] + 1          {penalty_p[p] is in local stable storage}
```

Algorithm 4.3: Basic algorithm: *UpdateLeader()* procedure.

become leader or not. This information will be used to avoid disagreement between correct and unstable processes. By definition, p is always well connected with itself (Lines 11-12). Also, p considers itself a candidate for becoming the leader (Line 13). Processes start the execution of the algorithm with no leader, which is reflected in the $\perp$ value assigned to the variable ℓ_p (Line 14). Finally, p initializes its timeout value with respect to each process q to η (the periodicity of message sending), and resets a timer to this value on q (Lines 15-17). After that, p starts the four tasks of the algorithm (Lines 18) concurrently .

Task 1. In Task 1 (Lines 19-22), which is executed periodically every η time units, process p first increments the sequence number $idMsgSent_p$ (Line 20), and then sends an *ALIVE* message to every other process q (Lines 21-22). Besides the sequence number, the message also includes p's incarnation value ($penalty_p[p]$), p's connectedness at reception with respect to q ($wellConnectedIn_p[q]$), and p's leader (ℓ_p). As we will see, the fact that $p = \ell_p$ will be interpreted by the receiver of this message as the indication that p considers itself as a candidate to become leader.

Task 2. In Task 2 (Lines 23-36), when process p receives an *ALIVE* message from another process q, p first updates if needed q's penalty value in $penalty_p[q]$ (Line 25), in which case $idMsgReceived_p[q]$ is set to the right value for the message to be processed in the subsequent code at Lines 27-28. Then, if the received message is the "*expected*" one from q, p resets $timer_p[q]$ to $Timeout_p[q]$ (Line 29), and updates its variables as follows: $idMsgReceived_p[q]$ is set to $idMsgSent_q$ (Line 30), $wellConnectedIn_p[q]$ is set to *TRUE* (Line 31), $wellConnectedOut_p[q]$ is set to $wellConnected_q$ (Line 32), and $leaderCandidates_p[q]$ is set to *TRUE* if $q = \ell_q$, and to *FALSE* otherwise (Line 33). Finally, p calls the *UpdateLeader()* procedure (Line 34) in order to determine its new leader. Finally, if the received message in this task is not the expected one from q but $penalty_q = penalty_p[q]$ (Line 35), then the message is discarded and p sends back a *PENDING* message to q (Line 36), in which p informs q about the penalty value and sequence number of the last expected message received from q ($penalty_p[q]$, $idMsgReceived_p[q]$).

Task 3. In Task 3 (Lines 37-40), when process p receives a *PENDING* message from another process q, if p's current penalty value corresponds to the value received in the message (Line 38), then p re-sends all the "pending" *ALIVE* messages between p and q (Lines 39-40), in order to re-establish the well connectedness between p and q. Pending messages correspond to the interval $idMsgReceived_q + 1, \ldots, idMsgSent_p$. This task allows coping with transient message omissions between p and q, as well as with messages that have not been received by q because it was down (i.e., crashed) when p sent them.

Task 4. In Task 4 (Lines 41-45), upon expiration of any of p's timers $timer_p[q]$, p first increments $Timeout_p[q]$ (Line 42) in order to avoid premature (erroneous) timeouts on q. However, it could be the case that q has really crashed, or has suffered a message omission with p. Hence, p's connectedness at reception with q is set to *FALSE* (Line 43), as well as $leaderCandidates_p[q]$ (Line 44). Then, p calls the *UpdateLeader()* procedure (Line 45) in order to determine its new leader.

UpdateLeader. In the *UpdateLeader()* procedure (Lines 46-55), p elects a leader only if it is well connected both at reception and at sending with a majority of processes (Line 51), selecting among the processes with which p is well connected the process q with smallest $penalty_p[q]$ value (using the process id to break ties). If p has $leaderCandidates_p[q]$ to *TRUE*, then p elects q as its leader (Line 53). Otherwise, i.e., if p is not well connected with a majority of processes, or the process q with smallest $penalty_p[q]$ value has $leaderCandidates_p[q]$ to *FALSE*, then p does not elect any leader, setting ℓ_p to $\perp$ (Line 50). Also, p increases its penalty value if it is not well connected with a majority of processes (Line 55).

4.5.1.3 Correctness of Algorithm

We now show the **correctness proof** for the *Basic Eventual Leader Service*, satisfying Definition 2, and the properties of Section 4.3.1.1.

All the time instants considered are assumed to be after every eventually up process has definitely recovered, and every eventually down process has definitely crashed and all its messages have been delivered, lost or omitted. Also, the unknown bounds on processing time and message communication delay apply to the set C (*CORE*), as well as the eventual reliability of communication links.

Lemma 1. *For every correct process p, i.e., belonging to the set C, eventually $penalty_p[p]$ does not increase anymore.*

Proof. Since p is correct, Line 3 gets executed a finite number of times at p. Observe that correct processes eventually communicate timely and stop omitting messages among them. By the algorithm there is a time after which correct processes stop erroneously suspecting each other, because they adjust their timeout values, and receive *ALIVE* messages before the timers expire. Since there is a majority of correct processes, eventually the condition of Line 51 will always be satisfied at p and Line 55 will not get executed anymore. □

Lemma 2. *For every incorrect process p, i.e., not belonging to the set C, eventually (1) p does not communicate without omission with a majority of processes anymore, or (2) $penalty_p[p]$ increases forever.*

Proof. Case (1) applies to eventually down processes. Case (2) applies to unstable processes, since Line 3 gets executed an infinite number of times. Finally, let p be an eventually up but incorrect process. By definition p will never communicate permanently without omission with a majority of processes (otherwise p would be a correct process). Hence, whenever p communicates without omission with a majority of processes, it will eventually "loose" this connectivity, and Line 55 will get executed. □

Remark. *Let ℓ be the correct process with smallest penalty value (using the process id to break ties).*

Lemma 3. *Eventually and permanently, $\ell_\ell = \ell$.*

Proof. By a similar reasoning to the one of Lemma 1, eventually ℓ will have $wellConnectedIn_\ell[q] = TRUE$ and $wellConnectedOut_\ell[q] = TRUE$ permanently for a majority of processes q. Consequently, in the *UpdateLeader()* procedure ℓ will select itself in Line 52 since it is the correct process with smallest penalty value. Since by definition every process p has $leaderCandidates_p[p] = TRUE$ permanently, ℓ will always set ℓ_ℓ to itself in Line 53 of the algorithm. □

Lemma 4. *Eventually, every ALIVE message sent by ℓ has $\ell_\ell = \ell$.*

Proof. Follows directly from Lemma 3 and Line 22 of the algorithm. □

Lemma 5. *Eventually and permanently, for every correct process p, $\ell_p = \ell$.*

Proof. Follows directly from Lemma 3 for $p = \ell$. Let $p \neq \ell$ be a correct process. By the same reasoning to Lemma 3, there is a time after which p will have $wellConnectedIn_p[q] = TRUE$ and $wellConnectedOut_p[q] = TRUE$ for a majority of processes q, and ℓ is in that majority. Consequently, in the *UpdateLeader()* procedure p will select ℓ as its leader since ℓ is the correct process with smallest penalty value, and by Lemma 4 all the *ALIVE* messages sent by ℓ have $\ell_\ell = \ell$. □

Lemma 6. *Eventually and permanently, every ALIVE message sent by every eventually up process $p \neq \ell$ has $\ell_p = \ell$ or $\ell_p = \perp$.*

Proof. Follows directly from Lemma 5 and Line 22 of the algorithm for every correct process $p \neq \ell$. Let p be an eventually up but incorrect process. By Lemma 2, eventually p does not communicate without omission with a majority of processes anymore, or $penalty_p[p]$ will be permanently bigger than the penalty value of any correct process. Hence, in case p communicates without omission with a majority of processes that includes ℓ, in the *UpdateLeader()* procedure p will select ℓ as its leader and will hence send *ALIVE* messages with $\ell_p = \ell$; otherwise, i.e., if p does not communicate without omission with a majority of processes anymore, or it communicates without omission with a majority of processes that does not include ℓ, then in the *UpdateLeader()* procedure p will set ℓ_p to $\perp$ and will hence send *ALIVE* messages with $\ell_p = \perp$. □

Lemma 7. *There is a time after which every incorrect process may only alternate between trusting either ℓ or no one.*

Proof. Follows directly from Lemma 6 for eventually up but incorrect processes. Let p be an unstable process. By the fact that p crashes and recovers an infinite number of times, eventually and permanently the penalty value of p will be bigger than the penalty value of any correct process. Also, by the algorithm, p initially sets ℓ_p to $\perp$. The only way for p to change its leader is to receive *ALIVE* messages timely from a majority of processes, which implies the reception from at least one correct process q. Hence, p will never select an incorrect process r as its leader in the *UpdateLeader()* procedure, since q's penalty value is smaller than r's. Moreover, since by Lemmas 4 and 6, among correct processes only ℓ sends *ALIVE* messages with $\ell_\ell = \ell$, the only way for p to change its leader is to receive *ALIVE* messages timely from a majority of processes that includes ℓ, in which case p will set ℓ as its leader. □

Theorem 1. *The algorithm presented in Algorithm 4.1, 4.2 and 4.3 implements Omega in crash-recovery and omissive systems, satisfying Definition 2.*

Proof. Follows directly from Lemmas 5 and 7. □

4.5.1.4 Constraints and Drawbacks

- The proposed algorithm does not address the unfavourable effects that present the selective omissions when creating scenarios where eventually up processes *connected* with a majority of processes do not communicate without omissions with the leader.
- The systematic use of stable storage can be very restrictive, regarding both space and latency.
- We understand better the relationship between different failure types and their net effect. However, it is possible to define a weaker model.

4.5.2 Communication-Efficient Eventual Leader Election

Our second proposal consists of a communication-efficient distributed leader election algorithm for crash-recovery and omission environments. The algorithm satisfies the definitions and properties presented in Section 4.3.1.

An eventual leader election service is communication-efficient when [LMA11]: *there is a time after which only one process (the elected leader) sends messages forever.* Observe that this definition implies that unstable processes eventually stop sending messages. Note also that in order to satisfy this property we require the elected leader to communicate *directly* without omissions with the rest of alive processes. Otherwise, i.e., if we weaken the communication assumption, message forwarding is required.

Formally, in this case eventually only the elected leader would send *new* messages forever, the rest of the processes would forward them to reach all other processes in the system. Therefore, the proposed algorithm is **communication-efficient**, i.e., eventually, each process of the well-connected majority communicates only with the elected leader.

It is worth mentioning that the considered system model is the one presented in Section 4.2, with the following complementary assumptions:

- The algorithm *does not assume the availability of stable storage* to store the number of times that the process suffers crash-recovery failures.
- Every process is *connected* with every other process by two *reliable communication links*, one in each direction.

4.5.2.1 Scenario of System

Algorithms 4.4, 4.5 and 4.6 show the proposal of a **Communication-Efficient Eventual Leader Election** in detail. Roughly speaking, as shown in Figure 4.3, every process p keeps track of its communication with every other process by exchanging *ALIVE* messages periodically. In the case that an omission is detected, communication in that link is marked as suspicious (in order to mend transient omissions, omitted messages are requested to be sent again).

Three types of messages are used: *RECOVERED* messages, which are sent during initialization and upon recovery, *ALIVE* messages, which are sent periodically and carry a sequence number in order to detect omissions, and *PENDING* messages, which are sent upon omission detection.

Note that eventually up and unstable processes send a finite and an infinite number of *RECOVERED* messages, respectively. For each unstable process u, we assume that some correct process receives an infinite subset of the RECOVERED messages sent by u.

4.5.2.2 The Algorithm

Communication-Efficient Eventual Leader Service is composed of an initialization part (Algorithm 4.4), five concurrent tasks (Algorithm 4.5) and a procedure for determining the leader (Algorithm 4.6), which we describe in the following paragraphs.

Initialization. During initialization, Lines 1 to 15, every process p starts the execution of the algorithm with no leader, which is reflected in the $\perp$ value assigned to the variable ℓ_p. Also, p sets a timer with respect to each process q. Figure 4.4 shows the initialization procedure executed by every process p when (i) p starts for the first time, and (ii) every time p recovers after a crash. This procedure initializes the set of variables that process p will use to track information about itself and the rest of processes in the system. It is worthy to note that ***stable storage is not*** used to store the number of crash-recovery failures.

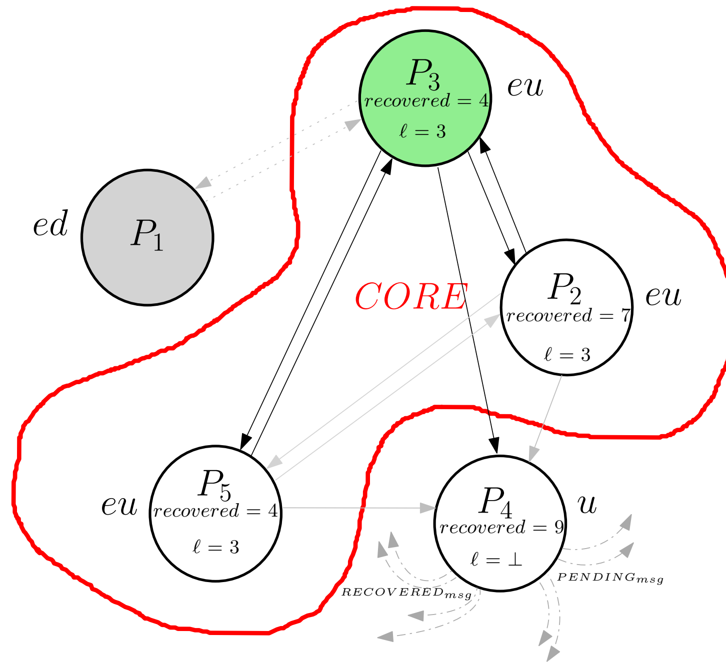


Figure 4.3: Communication-Efficient Eventual Leader Election (example)

```
1  || Initialization:                              {Executed by process p upon start/recovery}
2     recovered_p[p] ← 0
3     for all q ∈ Π except p do
4        recovered_p[q] ← 0                        {number of RECOVERED messages received from q}
5        send(RECOVERED, p) to q
6        idMsgSent_p[q] ← 0                        {sequence number for sending ALIVE messages to q}
7        idMsgReceived_p[q] ← 0                    {sequence number of the last ALIVE message received from q}
8        wellConnectedIn_p[q] ← FALSE              {TRUE if p communicates without omissions at reception with q}
9        wellConnectedOut_p[q] ← FALSE             {TRUE if p communicates without omissions at sending with q}
10       leaderCandidates_p[q] ← FALSE             {TRUE if q considers itself as leader}
11    ℓ_p ← ⊥                                      {initially p has no leader, i.e., it does not trust any process}
12    for all q ∈ Π except p do
13       Timeout_p[q] ← η                          {p's timeout on q is initialized to the periodicity of ALIVE sending by q in Task 2}
14       reset timer_p[q] to Timeout_p[q]
15    start tasks Task 1, Task 2, Task 3, Task 4 and Task 5
```

Algorithm 4.4: Communication-Efficient algorithm: process initialization.

16 || **Task 1:** When receive (*RECOVERED*, q) *{RECOVERED reception}*
17 $recovered_p[q] \leftarrow recovered_p[q] + 1$
18 UpdateLeader()

19 || **Task 2:** Repeat forever every η time units *{ALIVE sending}*
20 **if** $\ell_p = \bot$ **or** $\ell_p = p$ **then** *{if p has no leader or considers itself as leader, send ALIVE to all}*
21 **for** *all* $q \in \Pi$ *except* p **do**
22 $idMsgSent_p[q] \leftarrow idMsgSent_p[q] + 1$
23 send($ALIVE, p, recovered_p, idMsgSent_p[q], wellConnectedIn_p[q], \ell_p$) to q
24 **else** *{if p has a leader different from itself, send ALIVE to its leader}*
25 $idMsgSent_p[\ell_p] \leftarrow idMsgSent_p[\ell_p] + 1$
26 send($ALIVE, p, recovered_p, idMsgSent_p[\ell_p], wellConnectedIn_p[\ell_p], \ell_p$) to ℓ_p

27 || **Task 3:** When receive ($ALIVE, q, recovered_q, idMsgSent_q, wellConnected_q, \ell_q$) *{ALIVE reception}*
28 **if** $idMsgSent_q = idMsgReceived_p[q] + 1$ **then** *{if expected message}*
29 reset $timer_p[q]$ to $Timeout_p[q]$
30 **for** *all* $q \in \Pi$ **do**
31 $recovered_p[q] \leftarrow \max(recovered_p[q], recovered_q[q])$
32 $idMsgReceived_p[q] \leftarrow idMsgSent_q$
33 $wellConnectedIn_p[q] \leftarrow TRUE$ *{communication with q at reception is OK}*
34 $wellConnectedOut_p[q] \leftarrow wellConnected_q$ *{learn from q if p is well connected with q at sending}*
35 $leaderCandidates_p[q] \leftarrow (q = \ell_q)$ *{learn if q considers itself as a leader}*
36 UpdateLeader()
37 **else**
38 send($PENDING, p, idMsgReceived_p[q]$) to q *{ask for pending messages}*

39 || **Task 4:** When receive ($PENDING, q, idMsgReceived_q$) *{PENDING reception}*
40 **if** $idMsgReceived_q < idMsgSent_p[q]$ **then** *{if q has not received some of p's ALIVE messages, re-send them to q}*
41 **for** *all msgid from* $idMsgReceived_q + 1$ *to* $idMsgSent_p[q]$ **do**
42 send($ALIVE, p, recovered_p, msgid, wellConnectedIn_p[q], \ell_p$) to q
43 **else** *{if q's sequence number is bigger than p's (i.e., p has crashed and recovered), p adjusts its sequence number towards q}*
44 $idMsgSent_p[q] \leftarrow idMsgReceived_q$

45 || **Task 5:** Upon expiration of $Timer_p[q]$ *{timer expiration while waiting for the next ALIVE message}*
46 $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
47 $wellConnectedIn_p[q] \leftarrow FALSE$
48 $leaderCandidates_p[q] \leftarrow FALSE$
49 UpdateLeader()

Algorithm 4.5: Communication-Efficient algorithm: main tasks.

```
50 || Procedure UpdateLeader:                          {leader election based on collected information}
51 |  C_p ← {p}                                                   {C_p is p's estimation of the set CORE}
52 |  for all q ∈ Π except p do
53 |   |_ if wellConnectedIn_p[q] = TRUE and wellConnectedOut_p[q] = TRUE then insert q into C_p
54 |  ℓ_p ← ⊥
55 |  q ← select process ∈ C_p with smallest recovered_p[q], using process identifiers to break ties
56 |  if q = p then
57 |   | if |C_p| > n/2 then ℓ_p ← p
58 |  else
59 |   |_ if leaderCandidates_p[q] = TRUE then ℓ_p ← q
```

Algorithm 4.6: Communication-Efficient algorithm: *UpdateLeader()* procedure.

Task 1. In Task 1 (Lines 16-18), every process p accounts each *RECOVERED* message it receives, calling the *UpdateLeader()* procedure.

Task 2. In Task 2 (Lines 19-26), which is executed periodically, if either p has no leader or considers itself the leader, then it sends *ALIVE* to all processes. Otherwise, i.e., if p considers itself the leader, then it sends *ALIVE* to processes in C_p. Otherwise, if p has no leader, then it sends *ALIVE* to all processes. Finally, if p has a leader different from itself, then it sends *ALIVE* to its leader.

Task 3. In Task 3 (Lines 27-38), when p receives an *ALIVE* message from another process q, if the received message is the expected one from q, p resets $timer_p[q]$ and updates its variables accordingly, after which it calls the *UpdateLeader()* procedure. Otherwise, if the received message is not the expected one from q, then the message is discarded and p sends back a *PENDING* message to q.

Task 4. In Task 4 (Lines 39-44), when p receives a *PENDING* message from another process q, p either re-sends all the pending *ALIVE* messages between p and q or updates accordingly its sending sequence number with respect to q, (i.e., if p has crashed and recovered), in order to re-establish as soon as possible the well connectedness between them.

Task 5. In Task 5 (Lines 44-49), upon expiration of any of p's timers $timer_p[q]$, p first increments $Timeout_p[q]$ in order to avoid premature timeouts on q, updates its connectedness with q accordingly, and calls the *UpdateLeader()* procedure.

UpdateLeader. In the *UpdateLeader()* procedure, p first calculates the set C_p of processes with which it communicates well (including itself). Then, p selects the process q in C_p that has recovered fewer times. Finally, p sets q as its leader if either (1) $q = p$ and C_p contains a

majority of processes, or (2) $q \neq p$ and q considers itself as leader. Otherwise, no leader is set ($\perp$ value).

With this algorithm, the correct process ℓ with the lowest *RECOVERED* counter will be eventually and permanently elected as leader by all the correct processes. The rest of processes will adopt ℓ as their leader only if they communicate well with ℓ. Otherwise, they will have no leader. Hence, the algorithm satisfies Definition 2. Finally, observe that the algorithm is **communication-efficient**, i.e., eventually each process of the *well-connected* majority communicates only with the elected leader.

4.5.2.3 Correctness of Algorithm

We now show the **correctness proof** for the *Communication-Efficient Eventual Leader Service*, satisfying Definition 2 and the properties of Section 4.3.1.1. We also show that the algorithm is communication-efficient, i.e., eventually each process of the well-connected majority communicates only with the elected leader.

Although any process can suffer crash-recovery and omission failures, we assume that there is a time after which a majority of processes in the system remain up forever and stop omitting messages among them. Processes in this majority, named *CORE*, are said to be correct, while the rest of processes, i.e., eventually up processes not belonging to the set *CORE*, eventually down and unstable processes, are incorrect.

Observe that in the algorithm, *RECOVERED* messages are only sent during initialization (Line 5). Hence, by definition, eventually up processes send a finite number of *RECOVERED* messages, while unstable processes send an infinite number of *RECOVERED* messages. For each unstable process u, we assume that an infinite subset of the *RECOVERED* messages sent by u is received by some correct process.

All the time instants considered in the proof are assumed to be after:

(1) Every *eventually up process* has definitely recovered, and all its *RECOVERED* messages have already been either delivered or omitted, and

(2) every *eventually down process* has definitely crashed, and all its messages have already been either delivered or omitted.

Let be ℓ the correct process with smallest *RECOVERED* counter at any process in the system (using the process identifiers to break ties).

Lemma 8. *Eventually and permanently,* $\ell_\ell = \ell$.

Proof. Since there is a majority of correct processes *CORE* which includes ℓ such that eventually processes in the set *CORE* stop omitting messages among them, by the algorithm there is a time after which correct processes stop erroneously suspecting ℓ. This is because they adjust their

timeout value with respect to ℓ, and receive *ALIVE* messages timely, i.e., before the timer expires. Hence, process ℓ will have $wellConnectedIn_\ell[q] = TRUE$ and $wellConnectedOut_\ell[q] = TRUE$ for a majority of processes q. Consequently, in the *UpdateLeader()* procedure ℓ will select itself as leader since it is the correct process belonging to the set *CORE* with the smallest *RECOVERED* counter and $C_p > n/2$ (Lines 56-57). □

Lemma 9. *Eventually, every ALIVE message sent by* ℓ *has* $\ell_\ell = \ell$.

Proof. Follows directly from Lemma 8 and Line 23 of the algorithm. □

Lemma 10. *Eventually and permanently, for every correct process* p, $\ell_p = \ell$.

Proof. Follows directly from Lemma 8 for $p = \ell$. Let $p \neq \ell$ be a correct process. By Lemma 8 and Task 2 of the algorithm, ℓ sends an *ALIVE* message periodically to p. By the algorithm, eventually and permanently p will receive these *ALIVE* messages timely, i.e., before its timer with respect to ℓ expires. Consequently, in the *UpdateLeader()* procedure p will select ℓ as its leader since it is the process belonging to C_p with the smallest *RECOVERED* counter, and since by Lemma 9 all the *ALIVE* messages sent by ℓ have $\ell_\ell = \ell$, then $leaderCandidates_p[\ell] = TRUE$ (Line 59). □

Lemma 11. *Eventually and permanently, every ALIVE message sent by every eventually up process* $p \neq \ell$ *has* $\ell_p = \ell$ *or* $\ell_p = \perp$.

Proof. Follows directly from Lemma 10 and Line 26 of the algorithm for every correct process $p \neq \ell$. Let $p \neq \ell$ be an eventually up but incorrect process, i.e., not belonging to the set *CORE*. In case p communicates without omissions with ℓ, by the algorithm, eventually and permanently p will have $leaderCandidates_p[\ell] = TRUE$. Hence, in the *UpdateLeader()* procedure p will select ℓ as its leader and will hence send *ALIVE* messages with $\ell_p = \ell$; otherwise, i.e., if p does not communicate without omissions with ℓ, then in the *UpdateLeader()* procedure p will set ℓ_p to $\perp$ and will hence send *ALIVE* messages with $\ell_p = \perp$. □

Lemma 12. *There is a time after which every incorrect process* p *may only alternate between trusting either* $\perp$ *(i.e., it does not trust any process) or* ℓ.

Proof. Follows directly from Lemma 11 for eventually up but incorrect processes. Let p be an unstable process. By the fact that p crashes and recovers an infinite number of times, and sends a *RECOVERED* message upon recovery, eventually and permanently the *RECOVERED* counter of p at every correct process will be bigger than the *RECOVERED* counter of any eventually up process. Also, by the algorithm, p initially sets ℓ_p to $\perp$. By Lemmas 9 and 11, among eventually up processes only ℓ sends *ALIVE* messages with $\ell_\ell = \ell$. Hence, the only way for p to change its leader is to receive an *ALIVE* message from ℓ, in which case p will set ℓ as its leader in the *UpdateLeader()* procedure, since ℓ's *RECOVERED* counter at p is the smallest among all processes in C_p. □

Theorem 2. *The algorithm presented in Algorithm 4.4, 4.5 and 4.6 implements Omega in partially synchronous systems prone to crash-recovery and omission failures, satisfying Definition 2.*

Proof. Follows directly from Lemmas 10 and 12. ◻

Theorem 3. *The algorithm presented in Algorithm 4.4, 4.5 and 4.6 is communication-efficient, i.e., eventually each process of the well-connected majority communicates only with the elected leader.*

Proof. By Lemmas 9 and 10 and Task 2 (Algorithm 4.5), eventually and permanently among correct processes only ℓ sends a message periodically to the rest, while correct but non-leader processes send a message periodically to ℓ. ◻

4.5.2.4 Constraints and Drawbacks

- The proposed algorithm does not address the unfavourable effects that present the selective omissions when create scenarios where eventually up processes *connected* with a majority of processes do not communicate without omissions with the leader.
- We added an extra cost in handling messages, but stable storage is not necessary.

4.5.3 Indirect-Leader Trusting Mechanism

Our third proposal consists of a distributed leader election algorithm with an indirect leader trusting mechanism for crash-recovery and omission environments. The algorithm satisfies the definitions and properties presented in Section 4.3.1.

Unlike the previously presented algorithms, we provide an indirect-leader trusting mechanism that allows that even processes that suffer omissions with the leader may trust that leader, as long as they eventually communicate without omissions with a majority of correct processes. As a result, our approach allows implementing eventual leader election in systems with weaker assumptions compared to previous works.

The considered system model is the one presented in Section 4.2, augmented with the following complementary assumptions:

- The algorithm presented uses *stable storage* to store the number of times the process suffers crash-recovery failures.
- The communication channel between two processes is a *bidirectional reliable communication link*, and cannot create or alter messages.

4.5.3.1 Scenario of System

Algorithms 4.7, 4.8 and 4.9 show the proposal of an **Eventual Leader Election with Indirect-Leader Trusting Mechanism** in detail. Roughly speaking, as shown in Figure 4.4, processes exchange heartbeat messages periodically, messages carry a sequence number in order to detect omissions, and each process tracks: *(1) the number of times it recovers, and (2) the number of times they become disconnected from a majority.* The algorithm uses the rank concept (leadership-demoting), which represents the instability of every process in terms of computability and communication. We provide a mechanism to trust a leader even if communications with the leader are omission-prone, allowing to provide an eventual leader service that handles three main difficulties: *(a) eventual synchrony, (b) unstable processes due to crash-recovery failures and (c) communication failures due to selective omissions.*

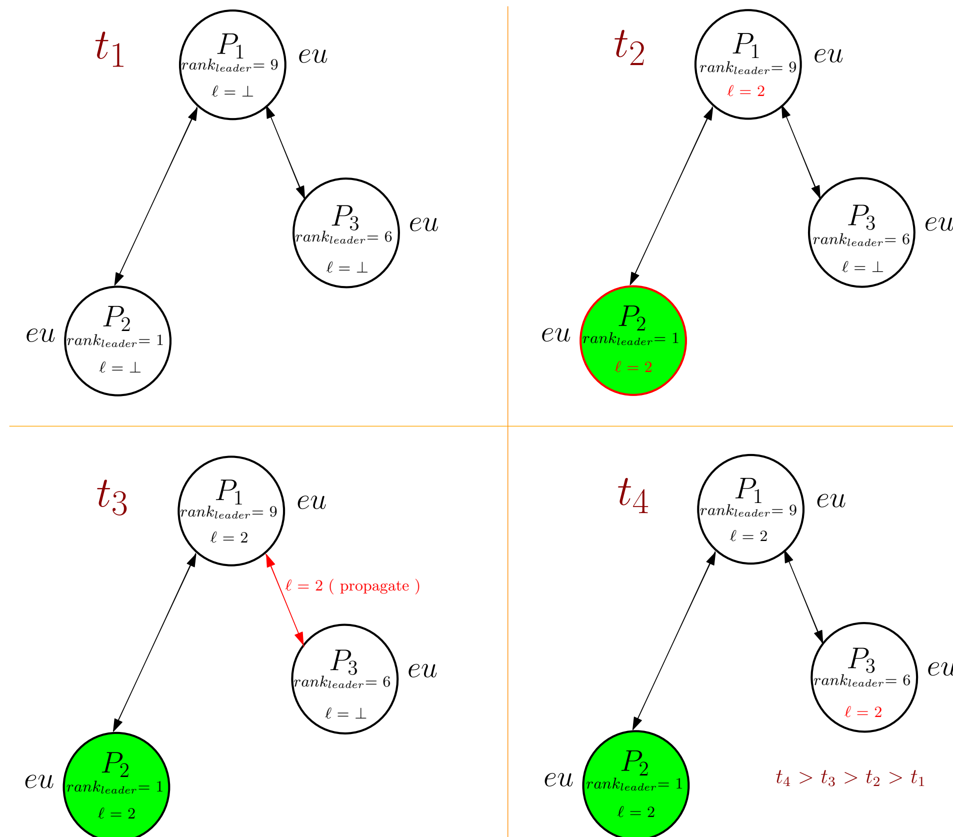


Figure 4.4: Eventual Leader Election with an Indirect-Leader Trusting Mechanism (example)

4.5.3.2 The Algorithm

Eventual Indirect-Leader Service is composed of an initialization part (Algorithm 4.7), four concurrent tasks (Algorithm 4.8), and a procedure for determining the leader (Algorithm 4.9), which we describe in the following paragraphs.

{*Note: leader is a complex variable* (id_ℓ, $epoch_\ell$, $disconns_\ell$, $lastAliveID_\ell$)}

```
1  || Task Initialization:                                {executed by every process p upon start/recovery}
2     if first execution of the algorithm then epoch_p ← 0
3     epoch_p ← epoch_p + 1                                {increment epoch after starting}
4     disconns_p ← 0                                       {initialize counter for disconnections from majority}
5     idMsgSent_p ← 0                                      {initialize message counter}
6     connectedWithMajority_p ← FALSE                      {initially, p is not connected to a majority}
7     NOLEADER ← (⊥, 0, 0)                                 {null leader (⊥, epochs, disconns)}
8     leaderTuple_p ← NOLEADER                             {output provided by the service}
9     leaderTupleToPropagate_p ← NOLEADER                  {leader propagated by p}
10    for all q ∈ Π except p do                            {initialize info about every other process}
11        Epochs_p[q] ← 0                                  {epoch of q}
12        IdMsgReceived_p[q] ← 0                           {last msg id received from q}
13        CommIn_p[q] ← FALSE                              {commun. state from q to p}
14        CommOut_p[q] ← FALSE                             {commun. state from p to q}
15        LeaderOthersTuple_p[q] ← NOLEADER
16        Timeouts_p[q] ← η
17        reset Timers_p[q] to Timeouts_p[q]
18    start tasks Task 1, Task 2, Task 3 and Task 4
```

Algorithm 4.7: Indirect-Leader Trusting Mechanism: process initialization.

INITIALIZATION. The initialization procedure is executed by every process p when: *(i) p starts up for the first time, and (ii) every time p recovers after a crash.* During initialization, Lines 1 to 18, this procedure initializes the set of variables that process p will use to track information about itself and the rest of processes in the system. It is worth to note that ***stable storage*** is used to store the value of the $epoch_p$ number despite crash-recovery failures. This variable contains the number of times a process has executed this procedure, i.e., it has recovered.

TASK 1. In Task 1 (Lines 19-22), every process p sends periodical *ALIVE* messages to every other process, so that process crashes can be detected. To detect message omissions, messages are tagged with a message identifier (variable $idMsgSent_p$), together with the epoch of the sender (variable $epoch_p$). Every process p monitors the communication *state* of its bidirectional links with every other process q, using variable $CommIn_p[q]$ for the communication from q to p and variable $CommOut_p[q]$ for the communication from p to q, being *TRUE* while all messages are delivered on time and *FALSE* otherwise. Observe that p can only monitor on reception the communication link from q to p, i.e., only one direction of the bidirectional link. For this reason, every process p includes in its *ALIVE* messages to process q the communication state from q to p. This way, every process gets information about the communication state of its outgoing communication links and, as result, of the bidirectional communication link between p and q. Finally, information about process leadership is sent into *ALIVE* messages (we will describe the content later).

19 || **Task 1:** Repeat forever every η time units *{send periodical ALIVEs}*
20   $idMsgSent_p \leftarrow idMsgSent_p + 1$
21   **for** *all* $q \in \Pi$ *except* p **do**
22   send($ALIVE$, p, $epoch_p$, $idMsgSent_p$, $CommIn_p[q]$, $leaderTupleToPropagate_p$) to q

23 || **Task 2:** When receive from q an $ALIVE$ message ($ALIVE$, q, $epoch_q$, $idMsgSent_q$, $Comm_q$, $leaderTuple_q$) *{ALIVE reception}*
24   **if** $epoch_q > Epochs_p[q]$ **then** *{if q has restarted since last message}*
25     $Epochs_p[q] \leftarrow epoch_q$
26     $IdMsgReceived_p[q] \leftarrow idMsgSent_q - 1$
27   **if** $epoch_q = Epochs_p[q]$ **then**
28   **if** $idMsgSent_q = IdMsgReceived_p[q] + 1$ **then** *{if expected message}*
29       $IdMsgReceived_p[q] \leftarrow idMsgSent_q$
30       $CommIn_p[q] \leftarrow TRUE$ *{communication from q is OK}*
31   reset $Timers_p[q]$ to $Timeouts_p[q]$
32       $CommOut_p[q] \leftarrow commun_q$ *{learn about comm. from p to q}*
33       $LeaderOthersTuple_p[q] \leftarrow leaderTuple_q$ *{learn from q its leader}*
34   UpdateLeader() *{process changes in leadership}*
35   **else**
36   send($PENDING$, p, $Epochs_p[q]$, $IdMsgReceived_p[q]$) to q

37 || **Task 3:** When receive ($PENDING$, q, $epoch_q$, $idMsgReceived_q$) *{PENDING reception}*
38   **if** $epoch_q = Epochs_p[q]$ **then**
39   **for** *all msgid from* $idMsgReceived_q + 1$ *to* $idMsgSent_p$ **do**
40   send($ALIVE$, p, $epoch_p$, $msgid_p$, $CommIn_p[q]$, $leaderTupleToPropagate_p$) to q

41 || **Task 4:** Upon expiration of $Timers_p[q]$ *{timer expiration while waiting for the next ALIVE message}*
42   $Timeouts_p[q] \leftarrow Timeouts_p[q] + 1$
43   $CommIn_p[q] \leftarrow FALSE$
44   $LeaderOthersTuple_p[q] \leftarrow NOLEADER$
45   UpdateLeader()

Algorithm 4.8: Indirect-Leader Trusting Mechanism: main tasks.

```
46 || Procedure UpdateLeader                                            {output a leader estimation}
47    C_p ← {p}                                              {First, calculate the set of connected processes}
48    for all q ∈ Π except p do
49        if ( CommIn_p[q] = TRUE and
               CommOut_p[q] = TRUE ) then insert q into C_p
      {Check if p has lost connectivity with a majority of processes                                    }
50    if |C_p| <= n/2 and connectedWithMajority_p = TRUE then
51        disconns_p = disconns_p + 1
52    connectedWithMajority_p ← (|C_p| > n/2)                                        {for next checking}
53    if connectedWithMajority_p = TRUE then                                     {Check if p can be leader}
54        tempLeaderTuple ← (p, epochs_p, disconns_p)
55    else
56        tempLeaderTuple ← NOLEADER
57    for all q ∈ C_p except p do                                            {Search for the best leader}
58        if LeaderOthersTuple_p[q].id ∉ {⊥, p} then
59            tempLeaderTuple = GetBestLeader(tempLeaderTuple, LeaderOthersTuple_p[q])
60    leaderTuple_p ← tempLeaderTuple_p                                         {Output of the service}
61    if leaderTuple_p.id ∈ C_p then                                {Leader propagation only when connected}
62        leaderTupleToPropagate_p = leaderTuple_p
63    else
64        leaderTupleToPropagate_p = NOLEADER

65 || Function GetBestLeader( leaderTuple_i, leaderTuple_j)                   {output the best candidate}
66    if leaderTuple_i.id = ⊥ then r
67    eturn leaderTuple_j
68    if leaderTuple_j.id = ⊥ then r
69    eturn leaderTuple_i
70    punish_i ← leaderTuple_i.epoch + leaderTuple_i.disconns
71    punish_j ← leaderTuple_j.epoch + leaderTuple_j.disconns
72    if ( punish_i < punish_j or
           (punish_i = punish_j and leaderTuple_i.id < leaderTuple_i.id) ) then
73        return leaderTuple_i
74    else
75        return leaderTuple_j
```

Algorithm 4.9: Indirect-Leader Trusting Mechanism: *UpdateLeader()* procedure.

Task 2, Task 3 and Task 4. Received *ALIVE* messages are processed in Task 2 (Lines 23-36). First, it is checked whether the received message is the expected one in the sequence (based on the epoch of the process and the last received message identifier) in order to control omissions. In case a gap in the sequence is detected, a *PENDING* message is sent to the sender reclaiming omitted messages (which will be processed by the sender process in Task 3, Lines 37-40). In case the received message is the expected one, communication state is set to *TRUE* (variable $CommIn_p[q]$) and information about process q is processed (variables $Comm_q$ and $LeaderTuple_q$). Finally, processing finishes by resetting the timer associated with the timely delivery of *ALIVE* messages from that process. Observe that this timer allows to control the timely delivery of *ALIVE* messages. If an expected message from a process q is not received before its associated timer expires, then process p will execute Task 4 (Lines 41-45), setting $CommIn_p[q]$ to *FALSE* and, thus, recalculating leadership.

Recall that every process p selects a leader based on its state and recollected information about the state of the processes p is *connected* with. A process l is considered leader if: *(i) l is* connected *with a majority of processes, and (ii) l has the lowest rank (based on the number of crash-recoveries and the number of disconnections from a majority).*

This implementation is presented in the procedure *UpdateLeader()* in Algorithm 4.9, which we will describe now. It is important to note that leader information is stored in a 3-tuple variable composed of:

(1) *identifier of the process*,

(2) *number of epochs*, and

(3) *number of disconnections*.

To distinguish this type of variables, the name of the variable includes the "*Tuple*" string, e.g., $leaderTuple_p$. To denote a null leader, i.e., no leader, we use a *NOLEADER* constant, composed of $(\perp, 0, 0)$ (see procedure *Initialization* in Algorithm 4.7). Each process p manages a $leaderTuple_p$ variable, which provides the output of the leader election service, i.e., $leader_p$. It also stores a vector named $LeaderOthersTuple_p$ containing leaders selected by other processes (according to the information extracted from received *ALIVE* messages). Process p is allowed to communicate its leader ℓ_p to other processes in case p and ℓ_p are *connected* (condition implemented by the variable $leaderTupleToPropagate_p$).

UpdateLeader. The *UpdateLeader* procedure in Algorithm 4.9 contains the core implementation of the leader election mechanism (Lines 46-64). Procedure starts in Line 47 by calculating the set of processes C_p process p is *connected* with (including itself). Then, the variable $disconns_p$ is incremented in case connectivity with a majority of processes has been lost (the variable $connectedWithMajority_p = TRUE$ indicates that process p was *connected* with a majority of processes). The next step, in Line 53, consist of searching for the best leader; first checking whether p itself is a leader candidate (see Line 53) and then evaluating the leader of

every process *connected* to p (see Line 57). A temporary variable *tempLeaderTuple* is used to avoid pointlessly changing the $leaderTuple_p$ variable. Observe that in Line 58, leaders of other processes are discarded when are either $\perp$ or p. The second case is due to the fact that p has more updated information about itself than any other process has.

GetBestLeader. Leader selection criteria is modularized by the *GetBestLeader* function (Lines 65-75), which analyses the information about two leaders and returns the best leader between both of them. In our implementation a rank/punish is calculated for each leader candidate based on its epoch and disconnection-from–majority counters. Finally, the function returns the leader with the smallest punish or, in case of draw, the leader with the smallest process identifier.

4.5.3.3 Correctness of Algorithm

We now show the **correctness proof** for the *Eventual Indirect-Leader Service*, satisfying Definition 2 and the properties of Section 4.3.1.1. All time instants considered in the proof are assumed to be after every eventually up process has definitely recovered, and every eventually down process has definitely crashed, and all its messages have been delivered, lost or omitted. Also, the unknown bounds on processing time and on message communication delay apply to the set *CORE* (C), as well as the eventual reliability of communication links.

Lemma 13. *$\forall p \in C$[8] eventually and permanently $punish_p$ will not change.*

Proof. $punish_p$ is a variable composed of two values: $epoch_p$ and $disconns_p$. We show that eventually both values will remain stable. Recall that by definition every process p in C is eventually up, which implies that eventually p will not recover again and, consequently, $epoch_p$ will not be updated any more (procedure *Initialization* in Figure 4.7). On the other hand, by definition p will eventually and permanently communicate without omissions with a majority of processes. Hence, eventually $disconns_p$ will not change any more (since Line 51 will not be executed). As a result, eventually $punish_p$ will not change any more for every $p \in C$. □

Lemma 14. *There exists a process $\ell \in C$ such that ℓ will eventually and permanently have the minimal punish value (and the lowest identifier in case of draw) among all processes in C.*

Proof. By Lemma 13 there exists a time after which $punish_p$ remains unchanged $\forall p \in C$. As a consequence, there will eventually and permanently exist a process $\ell \in C$ with the minimal $punish_\ell$ value forever (and smallest process identifier in case of draw) among all processes in C. □

Lemma 15. *$\forall q \notin C$ eventually and permanently either (1) q is eventually down or (2) $punish_q$ increases infinitely or (3) q is not* connected *with a majority of processes.*

[8]Recall the definition of the set C in Section 4.3.1.1

Proof. If $q \notin C$ then q is not eventually up or q is not eventually and permanently *connected* with a majority of eventually up. In case q is not eventually up, q is either an eventually down process, case (1) of the statement, or an unstable process. Unstable processes crash and recover infinite times, i.e., execute the *Initialization* procedure (Algorithm 4.7) infinite times. As a result, q will increment $epoch_p$ variable infinite times and, thus, $punish_p$ will be incremented forever, i.e., case (2) of the statement. On the other hand, in case q is an eventually up process that connects-and-disconnects from a majority forever, its $disconns_p$ variable will be incremented forever (see Line 52 in Figure 4.9) and, thus, $punish_p$ will be incremented forever, i.e., case (2) of the statement. Finally, in case q eventually and permanently is not *connected* with a majority, case (3) of the Lemma is fulfilled. □

Lemma 16. *Eventually and permanently* $\forall p \in \Pi$ connected *with a majority of processes* $punish_p \geq punish_\ell$.

Proof. By Lemma 14 $\forall p \in C$ eventually and permanently $punish_p \geq punish_\ell$. By Lemma 15, $\forall q \notin C$ and *connected* with a majority its variable $punish_q$ will be increased forever and, thus, eventually and permanently $punish_q > punish_\ell$. □

Remark. *A process p considers itself as leader, i.e., $leader_p = p$ in case (1) p is **connected** with a majority of processes and (2) p has the smallest **punish** value among the leader candidates p is **connected** with.*

Lemma 17. *Eventually and permanently* $leader_\ell = \ell$.

Proof. Since $\ell \in C$, eventually and permanently ℓ will be *connected* with a majority of processes. This way, when executing the *UpdateLeader()* procedure, ℓ will consider itself as a candidate to be leader (Line 54). By Lemma 16, ℓ has better *punish* than any other leader candidate q is compared with (See Lines 57–59), so $leader_\ell = \ell$. □

Lemma 18. $\forall p \in C$ *eventually and permanently* $leader_p = \ell$.

Proof. By Lemma 17 eventually and permanently $leader_\ell = \ell$. By Task 1 ℓ will spread its leadership to all processes and at least a majority of processes (recall that $\ell \in C$) will receive that information and consequently execute *UpdateLeader()*. By Lemma 16, eventually and permanently every process p connected with ℓ will have $leader_p = \ell$. Additionally, every process connected with ℓ is allowed to spread ℓ as leader (by using the $leaderTupleToPropagate_p$ variable), so that all processes in C can be reached. □

Lemma 19. $\forall q \notin C$ *eventually and permanently* $leader_q = \ell$ *or* $leader_q = \perp$.

Proof. In case q is connected with a process $p \in C$ then $leader_q = \ell$ (by Lemma 16). Consequently, every process $r \notin C$ *connected* with q will receive this information so eventually and permanently $leader_r = \ell$. In case q does not receive any information about leadership, $leader_q = \perp$. □

Theorem 4. *The algorithm presented in Algorithm 4.7, 4.8 and 4.9 implements a eventual leader election service (Omega in crash-recovery and omissive systems) satisfying Definition 2, and fulfills the properties of Section 4.3.1.1.*

Proof. Follows directly from Lemmas 18 and 19. □

4.5.3.4 Constraints and Drawbacks

- The systematic use of stable storage can be very restrictive, regarding both space and latency.
- System assumptions could be even weaker by incrementing the level of indirection when propagating leadership among processes (*n* hops).
- Other practical measures could be considered such as process connectivity or the quality of the majority of processes with which the analyzed process communicates well.

  their net effect. However, it is possible to achieve a weaker model yet.

4.6 Chapter Summary

We believe that an eventual leader election service exhibits significant advantages in dynamic and highly fault tolerance scenarios, such as those in asynchronous systems. The aforementioned service allows boosting distributed consensus algorithms, e.g., Paxos. We can summarize this chapter as follows:

- We provide a novel re-definition for the Omega failure detector based on a system model defined previously, in which a process can suffer both crash-recovery and omission failures at the same time, allowing some non-correct process to still agree on a common leader, and thus, participate in consensus.
- We provide the properties for an eventual leader service based on the new definition of the Omega failure detector in which processes can suffer both crash-recovery and omission failures at the same time.
- We show the implementation of three eventual leader election algorithms for crash-recovery and omission environments. The main difference between our algorithms and other similar algorithms of this type available in the literature consists of the failure handling strategy. Moreover, others handle crash and crash-recovery failure model, where the omissions are treated in the same way as a crash or crash-recovery failure, or omissions are not treated.

- *Regarding the eventual leader functionality.* We presented a novels leader election algorithm for crash-recovery and omission environments. The main novelty of the algorithm is that it tolerates the occurrence of both types of failures to any process, assuming that eventually, a majority of processes remain up and communicates without omissions. It eventually provides a common leader to every eventually up process able to eventually communicate without omissions with a majority of processes, or a null value otherwise.
- *Regarding the leadership capacity.* The effects of process failures have a significant impact on their leadership capacity. Initially, we only considered in this rank a punish value for every leadership-demoting failure type (crash-recovery, disconnection), which ensures that eventually unstable processes in terms of computability or communication will eventually be discarded as the leader. Additionally, some other (practical) measures could be considered, such as process connectivity or the quality of the majority of processes with which the analyzed process communicates well. For example, the works proposed in [AGSS13] for punishing too dynamic behavior of processes or in [BJS11] to assign scores dependent on the application.

5 Conclusions and Future Work

In this work, we have studied how to solve the leader election problem in fault-tolerant distributed systems with the purpose to solve the consensus problem. First, we have conducted a performance study of consensus algorithms in crash-recovery and general omission failure scenarios and, subsequently, we have proposed an eventual leader election service for distributed systems.

Solving the distributed leader election problem is not easy since we have to cope with: *(i) fault-tolerant asynchronous systems, and (ii) the crash-recovery and general omission failure models*. Accordingly, we have specified and implemented a distributed eventual leader election service on partially synchronous systems, in which processes can suffer both crash-recovery and omission failures at the same time.

In the following, we present the contributions of the work and identify directions for future research.

5.1 Research Assessment

We believe that the present work has contributed to the state of the art of the eventual leader election service in distributed systems in the following aspects:

Our experimental results have revealed that a leader election mechanism is more efficient than a rotating coordinator mechanism in order to solve the fault-tolerant distributed consensus problem.

> First, we have analyzed the building blocks of two well-known distributed consensus algorithms: *Chandra-Toueg [CT96] and Paxos [Lam98, PLL00, Lam01].* Both consensus algorithms have a comparable structure, i.e., they execute a sequence of rounds, each round being composed of four phases. Nevertheless, these algorithms present an inherent difference when having to determine who coordinates (*Chandra-Toueg*) or leads (*Paxos*) in a particular round. Chandra-Toueg is based on the *rotating coordinator mechanism*, while Paxos is based on a *leader election mechanism.* For this purpose, we conducted several executions of both algorithms on a partially synchronous model. To evaluate their efficiency, we have analyzed the *average early latency* and *average message complexity* in different failure scenarios. Our results exhibit that Paxos is more efficient than Chandra-Toueg when the process which coordinates or leads one round suffers a failure (crash,

omission, or crash-recovery). As a result, we can conclude that the leader election mechanism that uses Paxos provides a significant advantage in comparison with Chandra-Toueg's when running in faulty environments.

We have boosted the eventual leader election service as a component that enables the building of fault-tolerant distributed applications.

We have worked on a consensus algorithm based on an eventual leader election service since it allows to deal with more restricted scenarios regarding the computation and communication capabilities. Accordingly, an eventual leader election service may be used as a modular block to implement dependable distributed applications. Indeed, several fault-tolerant distributed agreement algorithms rely on an eventual leader election service in partially synchronous models. Besides, this service always preserves safety and guarantee liveness as soon as a unique leader remains for sufficiently long in the system [Gue00, GR04].

We have provided a novel definition of the Omega failure detector and the properties of a distributed eventual leader election service, which tolerates both crash-recovery and message omission failures at the same time.

Our proposal allows guaranteeing a distributed eventual leader election service that copes with false suspicions, unstable processes, and selective omissions. Such failures can affect the *computation* (*crash-recovery*) and *communication* (*message omissions*) between processes. Therefore, we have presented a novel definition of the Omega failure detector, denoted by $Omega_{crash-recovery,\ omission}$. It is interesting to note that the definition allows some incorrect processes still agree on a common leader. In this way, the processes can participate as much as possible in solving the distributed agreement problem, such as *the uniform-consensus problem.*

We have defined a weak system model of a distributed eventual leader election service.

We have presented a weak system model. This system model facilitates the specification and implementation of a distributed eventual leader election functionality. Such functionality enables affording to each process a distributed eventual leader election service on a partially synchronous model, wherein the processes can suffer failures of crash-recovery and omissions at the same time.

We have helped to better understand the relation between the different types of failures and, therefore, achieve a better view of their net effect on the system.

We have described the pitfalls of the defined system and how the proposal copes with them. We have provided a distributed eventual leader election service to manage three mainly difficulties; *eventual synchrony*, *unstable processes due to crash-recovery and omissions failures*, and *communication failures due to selective omissions.*

We have presented an approach of a progressive weakening in the implementation of a distributed eventual leader election service.

We have presented three novel mechanisms to eventually trust the same non-faulty process (the leader process), even if communications with the leader process are prone to crash-recovery and omission at the same time, as long as eventually there is a majority of eventually up processes that communicate without omissions with a majority of eventually up processes. The proposed implementations satisfy the definitions and properties previously specified. They are as follows:

(i) An *eventual leader election* algorithm, wherein any process may suffer failures forever, as long as a majority of processes meet some weak connectivity and reliability conditions.

(ii) A *communication-efficient leader election* algorithm, its communication pattern provides a flexible way to build overlay networks with a high ability of adaptation on different scenarios of failure.

(iii) An *eventual leader election with indirect leader trusting mechanism*, it allows that even processes that suffer omissions with the leader may trust that leader, as long as they eventually communicate without omissions with a majority of processes.

5.2 Future Work

We are interested in studying the weakest system model that allows implementing an eventual leader election in fault-tolerant distributed systems. The above will allow us to specify and implement new and novel services. Mainly, we are interested in working the following issues:

- *A performance study of the distributed eventual leader election service in wireless sensor network applications.*

  We believe that our algorithms can be used in devices that have a limited capacity of computation, storage, and battery (i.e., Distributed Sensor Networks). Therefore, we wish to carry out a new performance study of the proposed algorithms under different scenarios of failures. Through a practical comparison between the different implementations of the distributed eventual leader election service proposed in this work, analyzing its performance experimentally.

- *The chase of the weakest system model to implement a distributed eventual leader election service.*

  We seek to boost efficient communication in the different approaches proposed for an eventual leader election service. One way is improving the *indirect-leadership trusting mechanism*, avoiding cycles of outdated, propagated information, which could be implemented by using an additional timer for the leader. Additionally, incrementing the level of indirection when propagating leadership among processes. Additionally, other researching lines could be explored, such as addressing arbitrary failures, and different alternatives for membership in the processes of the system.

Bibliography

[AAH+85] Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors. *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, volume 190 of *Lecture Notes in Computer Science*. Springer, 1985.

[AAI+15] Sergio Arévalo, Antonio Fernández Anta, Damien Imbs, Ernesto Jiménez, and Michel Raynal. Failure detectors in homonymous distributed systems (with an application to consensus). *J. Parallel Distrib. Comput.*, 83:83–95, 2015.

[ACT97] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings*, volume 1320 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 1997.

[ACT99] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30, 1999.

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[ADFT04] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In Chaudhuri and Kutten [CK04], pages 328–337.

[ADFT08] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[AGSS13] Luciana Arantes, Fabíola Greve, Pierre Sens, and Véronique Simon. Eventual leader election in evolving mobile networks. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, volume 8304 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2013.

[AJR10] Antonio Fernández Anta, Ernesto Jiménez, and Michel Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *J. Comput. Sci. Technol.*, 25(6):1267–1281, 2010.

[And03] Ross J. Anderson. Cryptography and competition policy: issues with 'trusted computing'. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 3–10. ACM, 2003.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[ATD99] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. Revising the weakest failure detector for uniform reliable broadcast. In Jayanti [Jay99], pages 19–33.

[BBRP07] Roberto Baldoni, Marin Bertier, Michel Raynal, and Sara Tucci Piergiovanni. Looking for a definition of dynamic distributed systems. In Victor E. Malyshkin, editor, *Parallel Computing Technologies, 9th International Conference, PaCT 2007, Pereslavl-Zalessky, Russia, September 3-7, 2007, Proceedings*, volume 4671 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2007.

[BCT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In Babaoglu and Marzullo [BM96], pages 105–122.

[BFH+06] Zinaida Benenson, Felix C. Freiling, Thorsten Holz, Dogan Kesdogan, and Lucia Draque Penso. Safety, liveness, and information flow: Dependability revisited. In Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, and Erik Maehle, editors, *ARCS 2006 - 19th International Conference on Architecture of Computing Systems, Workshops Proceedings, March 16, 2006, Frankfurt am Main, Germany*, volume 81 of *LNI*, pages 56–65. GI, 2006.

[BHSS12] Fatemeh Borran, Martin Hutle, Nuno Santos, and André Schiper. Quantitative analysis of consensus algorithms. *IEEE Trans. Dependable Sec. Comput.*, 9(2):236–249, 2012.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[BJS11] Diogo Becker, Flavio Junqueira, and Marco Serafini. Leader election for replicated services using application scores. In Fabio Kon and Anne-Marie Kermarrec, editors, *Middleware 2011 - ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, volume 7049 of *Lecture Notes in Computer Science*, pages 289–308. Springer, 2011.

[BM93] Michael Barborak and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993.

[BM96] Özalp Babaoglu and Keith Marzullo, editors. *Distributed Algorithms, 10th International Workshop, WDAG '96, Bologna, Italy, October 9-11, 1996, Proceedings*, volume 1151 of *Lecture Notes in Computer Science*. Springer, 1996.

[Cas15] Arnaud Casteigts. Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks. In *Proceedings of the 8nd international ICST conference on simulation tools and techniques, SIMUTools'15, Athens, Greece*, 2015.

[CB01] Bernadette Charron-Bost. Agreement problems in fault-tolerant distributed systems. In *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics Piestany: Theory and Practice of Informatics*, SOFSEM '01, pages 10–32, London, UK, UK, 2001. Springer-Verlag.

[CBTB00] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 552–565. Springer, 2000.

[CFGA+12] Roberto Cortiñas, Felix C. Freiling, Marjan Ghajar-Azadanlou, Alberto Lafuente, Mikel Larrea, Lucia Draque Penso, and Iratxe Soraluze. Secure Failure Detection and Consensus in TrustedPals. *IEEE Trans. Dependable Sec. Comput.*, 9(4):610–625, 2012.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[CK04] Soma Chaudhuri and Shay Kutten, editors. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004.* ACM, 2004.

[CKV01] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[CRA14] C. Fernandez Campusano, R. Cortinas Rodriguez, and M. Larrea Alava. Boosting dependable ubiquitous computing: A case study. *IEEE Latin America Transactions*, 12(3):442–448, May 2014.

[Cri96] Flaviu Cristian. Synchronous and asynchronous group communication. *Commun. ACM*, 39(4):88–97, 1996.

[CS04] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1):15–37, 2004.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CTA02] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Computers*, 51(1):13–32, 2002.

[CUB02] Andrea Coccoli, Péter Urbán, and Andrea Bondavalli. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. In *DSN*, pages 551–560. IEEE Computer Society, 2002.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, January 1987.

[DFG+04] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In Chaudhuri and Kutten [CK04], pages 338–346.

[DGFF05] Carole Delporte-Gallet, Hugues Fauconnier, and Felix C. Freiling. Revisiting failure detection and consensus in omission failure environments. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2005.

[DGFG+11] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Anne-Marie Kermarrec, Eric Ruppert, and Hung Tran-The. Byzantine agreement with homonyms. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 21–30, New York, NY, USA, 2011. ACM.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[FCL13] Christian Fernández-Campusano, Roberto Cortiñas, and Mikel Larrea. Boosting dependable ubiquitous computing: A case study. In Gabriel Urzaiz, Sergio F. Ochoa, José Bravo, Liming Luke Chen, and Jonice Oliveira, editors, *Ubiquitous Computing and Ambient Intelligence. Context-Awareness and Context-Driven Interaction - 7th International Conference, UCAmI 2013, Carrillo, Costa Rica, December 2-6, 2013, Proceedings*, volume 8276 of *Lecture Notes in Computer Science*, pages 42–45. Springer, 2013.

[FCL14a] Christian Fernández-Campusano, Roberto Cortiñas, and Mikel Larrea. A leader election service for crash-recovery and omission environments. In Ramón Hervás, Sungyoung Lee, Chris D. Nugent, and José Bravo, editors, *Ubiquitous Computing and Ambient Intelligence. Personalisation and User Adapted Services - 8th International Conference, UCAmI 2014, Belfast, UK, December 2-5, 2014. Proceedings*, volume 8867 of *Lecture Notes in Computer Science*, pages 320–323. Springer, 2014.

[FCL14b] Christian Fernández-Campusano, Roberto Cortiñas, and Mikel Larrea. A performance study of consensus algorithms in omission and crash-recovery scenarios. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 240–243. IEEE Computer Society, 2014.

[FFP+06] Milan Fort, Felix C. Freiling, Lucia Draque Penso, Zinaida Benenson, and Dogan Kesdogan. Trustedpals: Secure multiparty computation implemented with smart cards. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2006.

[FGK11] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9, 2011.

[FLCR15] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. Eventual leader election despite crash-recovery and omission failures. In Guojun Wang, Tatsuhiro Tsuchiya, and Dong Xiang, editors, *21st IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2015, Zhangjiajie, China, November 18-20, 2015*, pages 209–214. IEEE Computer Society, 2015.

[FLCR16] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. A communication-efficient leader election algorithm in partially synchronous systems prone to crash-recovery and omission failures. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*, pages 8:1–8:4. ACM, 2016.

[FLCR17] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. A distributed leader election algorithm in crash-recovery and omissive systems. *Inf. Process. Lett.*, 118:100–104, 2017.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FSS05] Christof Fetzer, Ulrich Schmid, and Martin Süßkraut. On the possibility of consensus in asynchronous systems with finite average response times. In *25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA*, pages 271–280. IEEE Computer Society, 2005.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[GHM+99] Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, Rui Carlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In Sacha Krakowiak and Santosh K. Shrivastava, editors, *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, volume 1752 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1999.

[GL00] Eli Gafni and Leslie Lamport. Disk paxos. In Maurice Herlihy, editor, *DISC*, volume 1914 of *Lecture Notes in Computer Science*, pages 330–344. Springer, 2000.

[GL08] Rachid Guerraoui and Nancy A. Lynch. A general characterization of indulgence. *TAAS*, 3(4):20:1–20:19, 2008.

[GR04] Rachid Guerraoui and Michel Raynal. The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.

[GR06] Rachid Guerraoui and Michel Raynal. A leader election protocol for eventually synchronous shared memory systems. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems and the Second International Workshop on Collaborative Computing, Integration, and Assurance, SEUS 2006 / WCCIA 2006, Gyeongju, South Korea, April 27-28, 2006*, pages 75–80. IEEE Computer Society, 2006.

[Gue95] Rachid Guerraoui. Revistiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms, 9th International Workshop, WDAG '95, Le Mont-Saint-Michel, France, September 13-15, 1995, Proceedings*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 1995.

[Gue00] Rachid Guerraoui. Indulgent algorithms (preliminary version). In Gil Neiger, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, pages 289–297. ACM, 2000.

[Gue02] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[Gue08] Rachid Guerraoui. Failure detectors. In *Encyclopedia of Algorithms*. Springer-Verlag New York, 2008.

[Had90] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proceedings of the Asilomar Workshop on Fault-Tolerant Distributed Computing*, pages 201–208, London, UK, UK, 1990. Springer-Verlag.

[HMR98] Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *The Seventeenth Symposium on Reliable Distributed Systems, SRDS 1998, West Lafayette, Indiana, USA, October 20-22, 1998, Proceedings*, pages 280–286. IEEE Computer Society, 1998.

[HT93] Vassos Hadzilacos and Sam Toueg. Distributed systems (2nd ed.). In Sape Mullender, editor, *Distributed Systems*, chapter Fault-tolerant broadcasts and related problems, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[HUSK02] Naohiro Hayashibara, Péter Urbán, André Schiper, and Takuya Katayama. Performance comparison between the paxos and chandra-toueg consensus algorithms. In *Proc. of International Arab Conference on Information Technology, Doha, Qatar*, pages 526–533, 2002.

[HW05] Jean-François Hermant and Josef Widder. Implementing reliable distributed real-time systems with the *Theta*-model. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 334–350. Springer, 2005.

[JAT15] Ernesto Jiménez, Sergio Arévalo, and Jian Tang. Fault-tolerant broadcast in anonymous systems. *The Journal of Supercomputing*, 71(11):4172–4191, 2015.

[Jay99] Prasad Jayanti, editor. *Distributed Computing, 13th International Symposium, Bratislava, Slovak Republic, September 27-29, 1999, Proceedings*, volume 1693 of *Lecture Notes in Computer Science*. Springer, 1999.

[JT08] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 75–84. ACM, 2008.

[Küp13] Alptekin Küpçü. Distributing trusted third parties. *SIGACT News*, 44(2):92–112, 2013.

[LAA13] Mikel Larrea, Antonio Fernández Anta, and Sergio Arévalo. Implementing the weakest failure detector for solving the consensus problem. *IJPEDS*, 28(6):537–555, 2013.

[LAF99] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In Jayanti [Jay99], pages 34–48.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[Lam80] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, 1980.

[Lam96] Butler W. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo [BM96], pages 1–17.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam06a] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[Lam06b] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[Lap87] Jean-Claude Laprie. The dependability approach to critical computing systems. In Howard K. Nichols and Dan Simpson, editors, *ESEC '87, 1st European Software Engineering Conference, Strasbourg, France, September 9-11, 1987, Proceedings*, volume 289 of *Lecture Notes in Computer Science*, pages 233–243. Springer, 1987.

[Lap92] Jean-Claude Laprie. Dependability: A unifying concept for reliable, safe, secure computing. In Jan van Leeuwen, editor, *Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992*, volume A-12 of *IFIP Transactions*, pages 585–593. North-Holland, 1992.

[LFA00] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS*, pages 52–59, 2000.

[LFA04] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Computers*, 53(7):815–828, 2004.

[LM04] Leslie Lamport and Mike Massa. Cheap paxos. In *DSN*, pages 307–314. IEEE Computer Society, 2004.

[LMA11] Mikel Larrea, Cristian Martín, and Iratxe Soraluze Arriola. Communication-efficient leader election in crash-recovery systems. *Journal of Systems and Software*, 84(12):2186–2195, 2011.

[LMZ09] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Srikanta Tirthapura and Lorenzo Alvisi, editors, *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 312–313. ACM, 2009.

[LRAC12] Mikel Larrea, Michel Raynal, Iratxe Soraluze Arriola, and Roberto Cortiñas. Specifying and implementing an eventual leader service for dynamic systems. *IJWGS*, 8(3):204–224, 2012.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MA06] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, July 2006.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *SOSP*, pages 358–372. ACM, 2013.

[ML10] Cristian Martín and Mikel Larrea. A simple and communication-efficient omega algorithm in the crash-recovery model. *Inf. Process. Lett.*, 110(3):83–87, 2010.

[MLJ09] Cristian Martín, Mikel Larrea, and Ernesto Jiménez. Implementing the omega failure detector in the crash-recovery failure model. *J. Comput. Syst. Sci.*, 75(3):178–189, 2009.

[MMR02] Achour Mostéfaoui, Eric Mourgaya, and Michel Raynal. An introduction to oracles for asynchronous distributed systems. *Future Generation Comp. Syst.*, 18(6):757–767, 2002.

[MOZ05] Dahlia Malkhi, Florian Oprea, and Lidong Zhou. *Omega* meets paxos: Leader election and stability without eventual timely links. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2005.

[MR01] Achour Mostéfaoui and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[MRT+05] Achour Mostéfaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 109–118. IEEE Computer Society, 2005.

[MRT06] Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):656–666, 2006.

[OO14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319. USENIX Association, 2014.

[PLL00] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, 1986.

[Ray97] Michel Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *2nd High-Assurance Systems Engineering Workshop (HASE '97), August 11-12, 1997, Washington, DC, USA, Proceedings*, pages 209–214. IEEE Computer Society, 1997.

[Ray05] Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005.

[Ray07] Michel Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 12 - 14 July 2007, Cambridge, MA, USA*, pages 11–24. IEEE Computer Society, 2007.

[Ray10] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[Ray14] Michel Raynal. What can be computed in a distributed system? In Saddek Bensalem, Yassine Lakhnech, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, volume 8415 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2014.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch93] Fred B. Schneider. Distributed systems (2nd ed.). In Sape Mullender, editor, *Distributed Systems*, chapter What Good Are Models and What Models Are Good?, pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[SCL+11] Iratxe Soraluze, Roberto Cortiñas, Alberto Lafuente, Mikel Larrea, and Felix C. Freiling. Communication-efficient failure detection and consensus in omission environments. *Inf. Process. Lett.*, 111(6):262–268, 2011.

[SDS01] Nicole Sergent, Xavier Défago, and André Schiper. Impact of a failure detection mechanism on the performance of consensus. In *PRDC*, pages 137–145. IEEE Computer Society, 2001.

[SL84] Fred B. Schneider and Leslie Lamport. Paradigms for distributed programs. In Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors, *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, volume 190 of *Lecture Notes in Computer Science*, pages 431–480. Springer, 1984.

[SM95] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical report, Cornell University, Ithaca, NY, USA, 1995.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[UHSK04] Péter Urbán, Naohiro Hayashibara, André Schiper, and Takuya Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *SRDS*, pages 4–17. IEEE Computer Society, 2004.

[VR01] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

[ZWD+14] Lei Zou, Zidong Wang, Hongli Dong, Yurong Liu, and Huijun Gao. Time-and event-driven communication process for networked control systems: A survey. In *Abstract and Applied Analysis*, volume 2014. Hindawi Publishing Corporation, 2014.