UNIVERSIDAD DEL PAÍS VASCO

EUSKAL HERRIKO UNIBERTSITATEA

Departamento de Arquitectura y Tecnología de Computadores

Konputagailuen Arkitektura eta Teknologia Saila

Efficient Algorithms to Implement Failure Detectors and Solve Consensus in Distributed Systems

Memoria que para optar al grado de Doctor en Informática presenta

Mikel Larrea Álava

Dirigida por los Drs. André Schiper y Sergio Arévalo Viñuales

San Sebastián, julio de 2000

Abstract

The *Consensus* problem is a fundamental paradigm for fault-tolerant distributed computing. It abstracts a family of problems known as agreement problems, e.g., atomic broadcast, atomic commitment, group membership, and leader election. Any solution to Consensus can serve as a basic building block for solving such problems.

It is well known that Consensus cannot be solved deterministically in an asynchronous system that is subject to even a single process crash. This impossibility result stems from the lack of reliable failure detection in such a system. To circumvent such impossibility result, Chandra and Toueg introduced the concept of *unreliable failure detectors*, and showed how to use them to solve Consensus. Instead of giving specific *implementations*, they characterized failure detectors in terms of *abstract* properties.

This thesis focuses on the design of algorithms implementing failure detectors and its use to solve Consensus. We consider a partially synchronous system in which Consensus has been proven to be solvable. We first show that no one of the failure detector classes satisfying perpetual accuracy can be implemented in such a system. We then propose efficient algorithms implementing all the failure detector classes satisfying eventual accuracy. We also propose an optimal algorithm implementing $\diamond\mathcal{S}$, the weakest failure detector class for solving Consensus.

We also propose a new class of failure detectors, the Eventually Consistent class, denoted $\diamond\mathcal{C}$. This class adds an eventual leader election capability to the failure detection capability of other classes. We show that $\diamond\mathcal{C}$ is equivalent to $\diamond\mathcal{S}$. We also show that $\diamond\mathcal{C}$ can be implemented as efficiently as $\diamond\mathcal{S}$. We propose a Consensus algorithm that successfully exploits the leader election capability of $\diamond\mathcal{C}$ and performs better than all the previously proposed algorithms, which shows the potential of this new class of failure detectors.

Resumen

El *Consenso* es un problema fundamental en el contexto de los sistemas distribuidos tolerantes a fallos. Este problema sintetiza un conjunto de problemas conocidos como de "puesta de acuerdo", como por ejemplo el radiado fiable con orden total, el compromiso atómico, el problema de la membresía y la elección de líder. Cualquier solución al problema del Consenso puede ser utilizada como base para resolver estos problemas.

Es bien conocido que el Consenso no puede resolverse de manera determinista en un sistema asíncrono en el pueda haber fallos de proceso. Este resultado de imposibilidad se debe a que en un sistema asíncrono no es posible determinar de manera fiable si un proceso ha fallado o por el contrario únicamente es muy lento. Para superar esta imposibilidad, Chandra y Toueg propusieron el concepto de *detectores no fiables de fallos*, y mostraron cómo aplicarlos para resolver el Consenso. En lugar de proponer *implementaciones* concretas de detectores de fallos, los caracterizaron en base a propiedades *abstractas*.

Esta tesis se centra en el diseño de algoritmos que implementan detectores de fallos y su aplicación a la resolución del Consenso. Para ello se considera un sistema parcialmente síncrono en el que se ha demostrado que el Consenso puede ser resuelto. En primer lugar se demuestra que ninguna de las clases de detectores de fallos con precisión perpetua, esto es, desde el principio, puede ser implementada en dicho sistema. A continuación se proponen algoritmos eficientes que implementan todas las clases de detectores de fallos con precisión tarde o temprano. También se propone un algoritmo óptimo que implementa la clase de detectores de fallos más débil que permite resolver el Consenso, denominada $\diamond\mathcal{S}$.

Asimismo se propone una nueva clase de detectores de fallos, denominada "tarde o temprano consistente", $\diamond\mathcal{C}$. Esta clase añade a la capacidad de detectar fallos de otras clases la capacidad de – tarde o temprano – elegir un proceso líder. Se demuestra que las clases $\diamond\mathcal{C}$ y $\diamond\mathcal{S}$ son equivalentes. También se muestra que $\diamond\mathcal{C}$ se puede implementar tan eficientemente

como $\Diamond\mathcal{S}$. Para mostrar el potencial de esta nueva clase de detectores de fallos, se propone un algoritmo de Consenso que explota la capacidad de elegir un proceso líder de $\Diamond\mathcal{C}$, y que es más eficiente que todos los algoritmos propuestos con anterioridad.

To My Parents,

for their love and support

To Gurutze,

for countless things

Acknowledgements

Several people contributed either directly or indirectly to this thesis. First of all, I am very grateful to André Schiper and Sergio Arévalo, my advisors, for their support and guidance, specially during a time when this thesis was still in its beginnings. I would also like to express my gratitude to Alberto Lafuente for accepting to be my tutor in the Basque Country University.

A special thanks goes to Antonio Fernández Anta for his valuable contributions to this work. Many of his ideas have greatly improved the scientific quality of this research.

For the time and effort they have invested in judging the contents of this thesis, I would like to thank the members of the jury.

I would also like to thank the members of the Distributed Systems Group at the Public University of Navarra, the Distributed Operating Systems Group in Madrid, and the Operating Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne (specially Rachid Guerraoui), for their valuable comments and criticisms, and for motivating and encouraging my work.

I would like to acknowledge the Basque Government, Mondragon Eskola Politeknikoa, and the Public University of Navarra as the institutions that have provided the financial support for my research.

Last, but definitely not least, I am extremely thankful to my family for their love and support, and specially to my wife Gurutze, for her patience and understanding during these past months as well as for countless other things.

Contents

List of Figures

Chapter 1

Introduction

The increasing dependence of society on well-designed and well-functioning computer systems has led to an increasing demand for *dependable* systems. Dependability, i.e., reliability and availability, is one of the biggest trends in software technologies. In the past, it has been considered acceptable for services to be unavailable because of failures. This is rapidly changing: the requirement for high software reliability and availability is continually increasing in domains such as finance, booking-reservation, industrial control, telecommunication, etc.

The advent of low-cost, yet powerful workstations combined with fast and reliable networks has made the distributed execution of applications over a network a viable alternative to the use of one centralized large computer. A typical distributed system can be structured as a collection of processes, running on different hosts, working toward specified functional requirements. Thus, distributed computing systems are becoming more and more important nowadays. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. Thus, managing distributed systems is a difficult task since one must deal with remote communication and the various types of failures that may result from distribution.

One of the main advantages of distributed systems is its resilience to partial failures. Within a distributed system, services can be replicated on several computers, thus the failure of one computer does not affect the functioning of the system. This ability is of prime importance for the development of reliable systems.

1.1 Fault Tolerance in Distributed Systems

In distributed systems, *redundancy* and *agreement protocols* are the foundation for fault-tolerance [GS96a]. By replicating the critical components of the system, we make the whole system more reliable than his parts. Replication of components raises the question of how these replicas are coordinated. This is achieved via agreement protocols. Such agreement is necessary in order to guarantee consistency, e.g., to make a replicated server group behave as a single logical server.

Many agreement problems are related to the problem of *Consensus* [PSL80, Fis83, BDM93, CT96]. Consensus is a fundamental paradigm for fault-tolerant distributed computing. Informally, Consensus allows processes to reach a common decision, which depends on their initial inputs, despite failures. Consensus algorithms can be used as a basic building block to solve many other distributed agreement problems, such as atomic broadcast [Sch90, HT93], atomic commitment [Ske81, BHG87, Gue95], group membership [Mal96], and leader election [SM95]. Thus, Consensus can be seen as the "common denominator" of all these problems.

Given its wide applicability, Consensus has been thoroughly studied, and one important result by Fischer, Lynch and Paterson showed that Consensus cannot be solved deterministically in an asynchronous system with reliable channels that is subject to even a single *crash* failure [FLP85]. Informally, a distributed system is *asynchronous* if there is no bound on the transmission delay of messages or the relative speeds of processors. Essentially, the impossibility stems from the lack of reliable failure detection in such a system: in an asynchronous system, it is impossible to distinguish a failed process from a "very slow" one. This impossibility result has practical implications for distributed applications in which agreement is required, since it implies that no purely asynchronous algorithm can work correctly.

An elegant and modular approach to circumvent this impossibility result has been proposed by Chandra and Toueg [CT96], based on augmenting the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, they introduce the concept of *unreliable failure detectors* for systems with crash failures. The underlying idea is to reduce the asynchrony of the system by augmenting it with knowledge about failures (which may be imperfect), making it possible to solve many problems that are otherwise unsolvable. A failure detector is a service, usually local to each process, that provides information about process failures. It monitors a subset of the processes in the system, and maintains a list of those it currently suspects to have crashed. Some properties are defined on these lists, leading to different classes of failure detectors. Chandra and Toueg showed that Consensus can be solved with unreliable failure detectors, even if they make an infinite number of mistakes. They also showed that one of the failure detectors they defined is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes [CHT96].

Chandra and Toueg characterized failure detectors in terms of *abstract* properties as opposed to giving specific *implementations*. This approach allows to design algorithms and prove their correctness relying solely on these properties, without referring to low-level network parameters (such as the exact duration of the timeouts that are used to implement failure detectors).

1.2 About this Research

This thesis is about the design of fault-tolerant distributed algorithms. More precisely, we focus on the design of efficient and scalable algorithms that implement failure detectors, and its application to solve Consensus efficiently. Clearly, no failure detector that allows to solve Consensus can be implemented in a purely asynchronous system (this would contradict the impossibility result of Fischer, Lynch and Paterson). We consider a model of *partial synchrony* in which Consensus has been proven to be solvable.

1.2.1 Research Objectives

The primary goal of this work is to investigate the fundamental practical problem of how to implement failure detectors. Chandra and Toueg characterized failure detectors abstractly, in terms of *completeness* and *accuracy* properties. They only proposed an algorithm implementing one of the eight classes of failure detectors they defined. We study the design of more efficient algorithms implementing – if possible – all the classes of failure detectors defined in [CT96].

A secondary goal of this research is to better understand the failure detection abstraction for designing Consensus algorithms. Given an efficient algorithm implementing a failure detector, we study how it can be used to solve Consensus. We also investigate the definition of new classes of failure detectors that better capture the basic properties required to solve Consensus, allowing the design of efficient Consensus algorithms.

1.2.2 Research Contributions

This thesis provides five major contributions. In the context of the implementability of failure detectors in partially synchronous systems, we provide an impossibility result. In the context of the design of algorithms implementing failure detectors with eventual accuracy, we propose efficient ring-based algorithms and an optimal ordered-heartbeat algorithm. In the context of the failure detector classification, we propose a new class of failure detectors. Finally, in the context of Consensus algorithms based on failure detectors, we propose a new Consensus algorithm.

Impossibility result. We show that, among the eight classes of failure detectors defined by Chandra and Toueg in [CT96], no one of the four classes satisfying perpetual accuracy can be implemented in the model of partial synchrony considered in this thesis (even though Consensus is solvable in that model). Informally, this model stipulates that bounds on message delay and processor relative speeds exist but are unknown. Intuitively, the impossibility stems from the fact that perpetual accuracy requires that the failure detector does not make any mistake on at least one correct process, which cannot be achieved if the bounds are not known.

Ring-based algorithms. We propose a family of efficient algorithms that implement the four classes of failure detectors satisfying eventual accuracy. These algorithms are based on a ring arrangement of processes, and use a polling monitoring model, having a linear message complexity. The algorithms have been designed in a modular way. We first propose a basic algorithm providing only weak completeness (and no other property), an then we propose three extensions to this basic algorithm that provide eventual weak accuracy, eventual strong accuracy, and strong completeness respectively.

Ordered-heartbeat algorithm. We propose an optimal algorithm that implements $\diamond\mathcal{S}$, the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes. This algorithm uses a heartbeat monitoring model, and guarantees that eventually every correct process will permanently trust the same correct process. By simply suspecting all the processes except the trusted process, this algorithm satisfies the completeness and accuracy properties required by $\diamond\mathcal{S}$. We show that this algorithm performs better than previous algorithms. We also propose a new metric to evaluate the performance of algorithms implementing failure detectors, and show that the ordered-heartbeat algorithm is optimal with respect to this metric.

A new class of failure detectors. We propose a new class of failure detectors, the *Eventually Consistent* class, denoted $\diamond\mathcal{C}$. This class adds to the failure detection capability of other classes an eventual leader election capability. We show that $\diamond\mathcal{C}$ is equivalent to $\diamond\mathcal{S}$, the weakest class for solving Consensus. We also show that $\diamond\mathcal{C}$ can be implemented as efficiently as $\diamond\mathcal{S}$. In particular, we show that the two algorithms implementing $\diamond\mathcal{S}$ proposed in this thesis implement also $\diamond\mathcal{C}$.

A new Consensus algorithm. We propose a new Consensus algorithm that is based on the Eventually Consistent failure detector class. The algorithm relies strongly on the leader election capability of the class $\diamond\mathcal{C}$. To our knowledge, it is the first Consensus algorithm based on a failure detector satisfying eventual accuracy that does not rely on the rotating coordinator paradigm [CT96]. The algorithm, once the failure detector is stable, always solves Consensus in one single round, while algorithms based on the rotating coordinator paradigm may require several rounds. Thus, in this aspect the proposed algorithm is more efficient than previously proposed Consensus algorithms.

1.2.3 Thesis Organisation

The thesis is organised as follows. Chapter 2 discusses the different system models of distributed computing that have been considered in the literature, with special interest in models of partial synchrony and the model of unreliable failure detectors. Chapter 3 shows the impossibility of implementing any of the failure detector classes proposed by Chandra and Toueg that satisfy perpetual accuracy in a partially synchronous system in which Consensus has been proven to be solvable. Chapter 4 presents efficient algorithms implementing all the classes of failure detectors that satisfy eventual accuracy, and an optimal algorithm implementing the weakest failure detector for solving Consensus. Chapter 5 presents the *Eventually Consistent* class of failure detectors. It discusses the relation between this new class and existing classes, and presents algorithms that implement this new class. It also presents an efficient Consensus algorithm based on this class. Finally, Chapter 6 summarises the major results of this work and outlines future research directions.

$\infty\infty\infty$

Chapter 2

Distributed System Models

2.1 Introduction

When describing any kind of system, it is useful to sate clearly which assumptions have been considered about it, in other words, to state the system model. This chapter discusses the different system models that have been proposed in the area of fault-tolerant distributed computing, some of which are used in this thesis. We also present existing important results for some models.

First of all, this thesis is focused on *message-passing* distributed systems, in which processes communicate only by sending and receiving messages over a communication network (we do not consider any distributed shared-memory mechanism). The two basic abstractions that characterize such a distributed system are *processes* and *communication channels*.

2.2 Processes and Communication Channels

A typical distributed system consists of a set of processes exchanging messages via communication channels. A process can either be correct or incorrect. Correct processes behave accordingly to their specification, while incorrect processes fail. Depending on how a process behaves when it fails, failures can be classified as follows (the list is not exhaustive):

- **Crash.** Failure that causes the process to stop its execution. Up to the point of the crash the process executes correctly, and after the crash it is completely inactive.

- **Omission.** Failure that causes the process to fail to receive or to send a message.

- **Byzantine.** Arbitrary failure which causes the process to behave in a totally arbitrary, even malevolent, manner. In particular, it can send messages when it is not supposed to, act crashed for awhile and then revive itself, etc.

Crash failures are a particular case of omission failures, while Byzantine failures include all types of failures (see Figure 2.1).

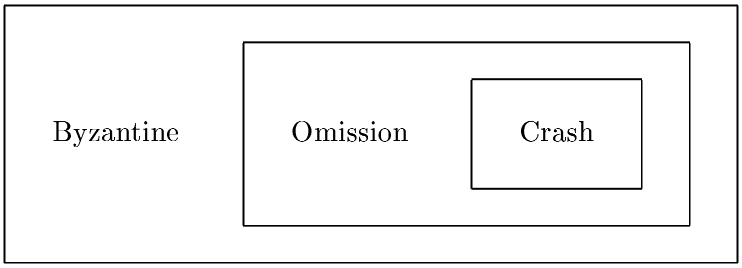

Figure 2.1: Failure classification.

In this thesis, we consider the following process model:

- The system is composed by a finite set of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$. The set of processes does not change during the execution of the system.

- Processes can only fail by *crashing*, that is, by prematurely halting. We do not consider omission or Byzantine failures. Moreover, crashes are *permanent*, i.e., crashed processes do not recover. We consider that a process is *up* at time t if it is not crashed at time t. Similarly, we consider that a process is *down* at time t if it is crashed at time t. A *correct* process is a process that is always up. Similarly, an *incorrect* process is a process that is eventually – and permanently – down.

In every execution of the system we identify two complementary subsets of Π: the subset of processes that do not fail, denoted $correct$, and the subset of processes that do fail, denoted $crashed$. We use $\mathcal{C}$ to denote the number of correct processes in the system, which we assume is at least one, i.e., $\mathcal{C} = |correct| > 0$.

With respect to the communication channel model, we consider *reliable* channels. We assume that processes are fully connected, i.e., every pair of processes is connected by a reliable point-to-point communication channel. A reliable channel is defined as follows:

- **Reliable channel.** If a process p sends a message m to a correct process q, then q eventually receives m.

Usually, reliable channels are implemented using retransmission techniques (the message is retransmitted by the sender until the receiver, which we assume is correct, acknowledges it). The potential crash of the sender makes difficult the implementation of such a channel model. In practice, distributed algorithms are usually designed based on a weaker type of channel, called *quasi-reliable* channel[1], which is defined as follows:

- **Quasi-reliable channel.** If a *correct* process p sends a message m to a correct process q, then q eventually receives m.

The algorithms presented in Chapters 4 and 5 assume quasi-reliable channels. On the other hand, the impossibility result of Chapter 3 is proven assuming reliable channels[2].

2.3 Asynchronous vs. Synchronous System Model

According to synchronisation aspects of processes and communication channels, there are two extreme system models: the asynchronous model and the synchronous model.

In the *synchronous* model there are known upper bounds Δ and Φ on the transmission delay of messages and the relative speeds of processes (i.e., when the slowest process performs *one* step, the fastest process performs at most Φ steps). It is also assumed that processes have synchronized clocks [HT94]. Agreement problems, e.g., Consensus, can be solved in synchronous systems [FM90, Zam96]. Intuitively, this is because a synchronous system allows to reliably detect the crash of other processes using timeouts.

A drawback of the synchronous model is that it requires to be conservative when defining the bounds on the transmission delay of messages and the relative speeds of processes. Otherwise the synchrony assumptions can be violated. Thus, a major drawback of algorithms based on the synchronous model is that, whenever an execution of the system violates the assumptions of the system model, the algorithm may loose correctness (i.e., violate its safety properties).

[1]Also *eventual reliable* channel, or *correct-restricted* reliable channel.
[2]Impossibility results are usually proven in the stronger, i.e., reliable, channel model.

On the other hand, in the *asynchronous* model there are no upper bounds on neither the transmission delay of messages nor the relative speeds of processes. Thus, to say that a system is asynchronous is to make no timing assumptions. This model is attractive because of its simplicity, and because the solutions valid for it are also valid for virtually any real system[3]. Moreover, in practice, it is very rare to find a system which always behaves as a synchronous system. Usually, on the contrary, unexpected loads turn a system into asynchronous. Thus, the asynchronous model adequately models an unpredictable load of processes and communication channels [CT96, GS97].

Another advantage of using the asynchronous model is that any algorithm designed under its assumptions preserves its correctness (safety properties) in any other system model.

However, for completely asynchronous distributed systems, it has been shown by Fischer, Lynch and Paterson that no agreement among processes can be solved in a deterministic and fault-tolerant way [FLP85]. More precisely, this paper shows that Consensus cannot be solved deterministically in an asynchronous system with reliable channels that is subject to even a single crash failure. Intuitively, this is because, in such a system, it is impossible to distinguish a very slow process from one that has actually crashed. As a consequence, solutions to many problems that require *all* processes to participate in a nontrivial way are impossible [FLP85, SM95, CHTCB96, Ske81, BT93, Gue95].

In order to overcome the so-called FLP impossibility result, different approaches have been considered in the literature: the use of randomization techniques [BO83, Rab83, Tou84, Bra87, CD89], the definition of problems weaker than Consensus and their solutions [DLP+86, Cha93b], the definition of several models of *partial synchrony* [DDS87, DLS88], and the model of *unreliable failure detectors* [CT96, CHT96]. Given its important relation with this research, the last two approaches, i.e., partial synchrony and unreliable failure detectors, will be presented in the following two sections.

2.4 Partial Synchrony

One of the approaches proposed to circumvent the unsolvability of Consensus in asynchronous systems is based on the observation that the completely synchronous and completely asyn-

[3]Clearly, the asynchronous model can model any *real* distributed system.

chronous models of distributed systems are the two ends of a large range of models, and that between them there lie a variety of intermediate "partially synchronous" models.

In particular, Dolev et al. [DDS87] defined 32 models of partial synchrony by considering five key parameters, each of which can be set to "favorable" or "unfavorable". For instance, one of the parameters is the maximum message delay, which can be bounded and known (favorable setting) or unbounded (unfavorable setting). Each of the 32 models corresponds to a particular setting of the 5 parameters. Dolev et al. [DDS87] identify four "minimal" models in which Consensus is solvable. These are minimal in the sense that the weakening of any parameter for favorable to unfavorable would yield a model of partial synchrony where Consensus is unsolvable. In particular, they show that the existence of an upper bound Δ on the time for messages to be delivered and an upper bound Φ on the relative speeds of processors is necessary to solve Consensus. If either an upper bound on message delivery time does not exist or an upper bound on relative processor speeds does not exist, then there is no Consensus protocol resilient to even one crash failure [DLS88].

Dwork et al. [DLS88] consider the following two models of partial synchrony:

- M_1: in every execution, there are bounds on relative process speeds and on message transmission times, but these bounds are not known.

- M_2: bounds exist and are known, but they hold only after some unknown (but finite) time (called GST for *Global Stabilization Time*).

A system that conforms to model M_2 can be seen as asynchronous up to GST, and as synchronous after GST. Thus, M_2 can be seen as an *Eventually Synchronous* model. However, it is important to note that the actual value of GST is not known.

From the point of view of the solvability of Consensus, both models M_1 and M_2 satisfy the requirement of the existence of the bounds on message transmission times and relative process speeds. Hence, Consensus is solvable in both models. Dwork et al. [DLS88] propose Consensus algorithms for various fault models[4] that work correctly regardless of the actual values of the bounds (in the case of model M_1), and the actual value of GST (in the case of model M_2).

[4]Model M_2 becomes interesting if channels are unreliable before GST. In this case, algorithms must include techniques to mask the loss of messages.

In [CT96], Chandra and Toueg propose a weaker model of partial synchrony M_3, which generalizes the two previous models M_1 and M_2:

- M_3: bounds exist, but they are not known *and* they hold only after some unknown (but finite) time GST.

A system that conforms to model M_3 can be seen as asynchronous up to time GST, and as a system conforming to M_1 after GST. Note also that every system that conforms to M_1 or M_2 also conforms to M_3. Moreover, if we consider that communication channels are reliable, then models M_3 and M_1 are equivalent, i.e., every system that conforms to M_3 also conforms to M_1 (with possibly different values for the unknown bounds). On the other hand, if communication channels are not reliable up to GST, then model M_3 is *strictly* weaker than M_1: not every system that conforms to M_3 conforms to M_1. Consider for example an execution of a system that conforms to M_3 in which some message m sent before GST is lost. Clearly, there is no upper bound on m's transmission delay, and thus this system does not conform to M_1. In the case of reliable channels, m will eventually be received (maybe after GST), and thus an upper bound exists.

In this thesis, we focus on the models of partial synchrony in which the bounds are not known. In particular, we will consider the partially synchronous model M_3[5].

The models of partial synchrony M_1, M_2, and M_3 state that the existing bounds must hold *permanently* (in the case of models M_2 and M_3, after GST). However, in practice, it is not really required that this condition holds forever. When solving a problem that "terminates", e.g., Consensus, it is enough that the bounds hold for "sufficiently long" periods of time. However, from a theoretical point of view it is convenient to state the properties in the stronger form given above.

2.5 Unreliable Failure Detectors

An alternative and elegant approach to circumvent the unsolvability of Consensus in asynchronous systems has been proposed by Chandra and Toueg [CT96], based on the idea of

[5]Actually, the model M_3 will be considered in Chapters 3 and 4. In Chapter 5, we will consider the asynchronous model *augmented* with a failure detector such that Consensus is solvable.

augmenting the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, they model the concept of *unreliable failure detectors* for systems with *crash* failures.

Informally, an unreliable failure detector is a distributed "oracle" that gives (possibly incorrect) hints about which processes may have crashed so far. Each process has access to a local *failure detector module* that monitors other processes in the system, and maintains a set[6] of those that it currently suspects to have crashed. A failure detector module can make mistakes by not suspecting a crashed process or by erroneously adding processes to its set of suspects, i.e., it can suspect that a process p has crashed even though p is still running. If it later believes that suspecting p was a mistake, it can remove p from its set of suspects. Thus, each module may repeatedly add and remove processes from its set of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different sets of suspects.

A *class* of failure detectors is characterized by specifying the *completeness* and *accuracy* properties that failure detectors in this class must satisfy. Roughly speaking, *completeness* requires that a failure detector eventually suspects every process that actually crashes,[7] while *accuracy* restricts the mistakes (i.e., false suspicions) that a failure detector can make. Chandra and Toueg defined two completeness and four accuracy properties, which combined give rise to eight classes of failure detectors. They also showed that Consensus can be solved with any of these classes.

2.5.1 Properties and Classification

Chandra and Toueg defined two completeness properties and four accuracy properties that a failure detector may satisfy. Considering completeness, we have the following two properties:

- *Strong Completeness.* Eventually every process that crashes is permanently suspected by *every* correct process.

- *Weak Completeness.* Eventually every process that crashes is permanently suspected by *some* correct process.

[6] It is also common to refer this set as a *list* of suspects.

[7] We say that the failure detector suspects that a process p has crashed if *any* local failure detector module suspects that p has crashed.

Note that with weak completeness the information that a process has crashed is somewhere in the system, but it is unknown where.

Completeness by itself is not a useful property. For example, strong completeness can be trivially satisfied by forcing every process to permanently suspect every other process in the system. Such a failure detector is clearly useless, since it provides no information about failures. To be useful, a failure detector must also satisfy some accuracy property that restricts the *mistakes* that it can make. Chandra and Toueg consider the following four accuracy properties:

- *Perpetual Strong Accuracy.* No process is suspected before it crashes.
- *Perpetual Weak Accuracy.* Some correct process is never suspected.
- *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 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. As it will be shown in Chapter 3, this type of accuracy cannot be achieved in the partially synchronous model considered in this thesis.

Combining one of the two completeness properties with one of the four accuracy properties we obtain a *class* of failure detectors. There are eight different classes, which are presented in Figure 2.2. Out of these, Chandra et al. showed in [CHT96] that $\Diamond\mathcal{W}$ is the weakest class of failure detectors required for solving Consensus. They also showed in [CT96] that the failure detector classes $\Diamond\mathcal{S}$ and $\Diamond\mathcal{W}$ are actually equivalent, i.e., one can be transformed into the other and vice versa. Transforming $\Diamond\mathcal{S}$ into $\Diamond\mathcal{W}$ is trivial, since $\Diamond\mathcal{S}$ is a subclass of $\Diamond\mathcal{W}$. To transform $\Diamond\mathcal{W}$ into $\Diamond\mathcal{S}$, they propose in [CT96] a distributed algorithm that transforms weak completeness into strong completeness, preserving accuracy. The transformation algorithm is given in Appendix A.

Clearly, the stronger a failure detector is, the simpler the algorithm based on it will be. For example, an algorithm that solves Consensus is much simpler if we assume a failure detector of class $\diamond\mathcal{S}$ than if we assume a failure detector of class $\diamond\mathcal{W}$[8]. Clearly, the second algorithm must cope with the lack of completeness of $\diamond\mathcal{W}$ with respect to $\diamond\mathcal{S}$.

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	*Quasi-Perfect* $\mathcal{Q}$	*Weak* $\mathcal{W}$	*Eventually Quasi-Perfect* $\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Figure 2.2: Eight classes of failure detectors defined in terms of completeness and accuracy.

2.5.2 Solving Consensus using Unreliable Failure Detectors

Chandra and Toueg proposed in [CT96] a Consensus algorithm based on a failure detector of class $\diamond\mathcal{S}$. This algorithm assumes a majority of correct process (as it is shown in [CT96], this is a necessary requirement for solving Consensus using a failure detector with eventual accuracy, even $\diamond\mathcal{P}$). The algorithm proceeds in asynchronous rounds, in each of which a different process acts as coordinator. This approach is known as the *rotating coordinator paradigm*. If the coordinator of a round is not correct or is suspected by several processes the round may fail and the consensus is not reached in that round. With a failure detector of class $\diamond\mathcal{S}$ it is guaranteed the existence of a correct process p that is eventually not suspected by any correct process. If eventually p becomes the coordinator of a round after no process suspects it, the consensus is guaranteed to be reached. The detailed algorithm is given in Appendix B.

Several other algorithms that solve Consensus using $\diamond\mathcal{S}$ have been recently proposed [Sch97, HR99, MR99a, HMR00]. Basically, they differ from the algorithm of Chandra and Toueg in the message exchange pattern between processes in each round: in the case of Chandra-Toueg's algorithm, it is centralized on the coordinator of the round, while in the rest of algorithms it is (or it can be) decentralized. Nevertheless, they all rely on the rotating coordinator paradigm.

A good feature of Consensus algorithms based on $\diamond\mathcal{S}$ is that they are *safe*, i.e., they

[8] Actually, no algorithm for solving Consensus using *only* $\diamond\mathcal{W}$ has ever been proposed.

preserve safety even if neither of strong completeness nor eventual weak accuracy is satisfied. In particular, even if there is a crash that no process ever detects, and all correct processes are repeatedly (and forever) falsely suspected, (1) no two processes ever disagree on a decision and (2) no process ever decides on a value that was not proposed by some process.

2.5.3 Failure Detectors and Partial Synchrony

Failure detectors can be viewed as an abstract way of incorporating partial synchrony assumptions into the model of computation [CT96]. Instead of focusing on the *timing assumptions* of a given model of partial synchrony, the model of failure detectors considers *abstract properties* that must be satisfied in order to solve Consensus. Clearly, the failure detector encapsulates the timing assumptions. With this approach, an agreement algorithm does not need to directly handle timing parameters. It just interacts with the failure detector through a simple interface[9]. This leads to the practical problem of *implementing* a given failure detector in a specific model of partial synchrony.

From the FLP impossibility result [FLP85] and the possibility of solving Consensus using unreliable failure detectors [CT96], it can be derived the impossibility of, in a purely asynchronous system, implementing any of Chandra-Toueg's classes of failure detectors[10]. On the other hand, in a synchronous system even a *Perfect* failure detector, i.e., one that does not make mistakes, can be implemented. In such a system, one can use the known upper bounds on message delay and processor relative speeds to build a simple timeout-based algorithm that reliably detects the failure of processes.

Finally, it has been shown that Consensus can be solved in several models of partial synchrony [DDS87, DLS88]. The Consensus algorithms proposed in these papers actually have some built-in failure detection mechanism. This failure detection mechanism does not need to be completely reliable, but has to be at least as strong as $\diamond\mathcal{W}$, the weakest failure detector for solving Consensus. For example, in the models of partial synchrony defined by Dwork et al. [DLS88] and Chandra and Toueg [CT96], it is possible to implement a failure detector that satisfies the properties of $\diamond\mathcal{W}$, even those of $\diamond\mathcal{P}$. In this thesis, we have

[9] Models using unreliable failure detectors are no longer truly asynchronous; they merely produce the illusion of an asynchronous system by encapsulating all references to time in the failure detector.

[10] Clearly, such an implementation could be used to solve Consensus in an asynchronous system, contradicting the FLP impossibility result.

investigated which classes of failure detectors, among those of Figure 2.2, can be implemented in such a partially synchronous system.

Figure 2.3 shows a general architecture of a fault-tolerant Consensus algorithm based on a failure detector. If we assume the existence of various communication subsystems, only the communication subsystem that supports the implementation of the failure detector has to be strictly stronger than the asynchronous system (in order to implement any of the classes of failure detectors that allow to solve Consensus). The part of the Consensus algorithm that is not related to the failure detector[11] could use a completely asynchronous – but reliable – communication subsystem.

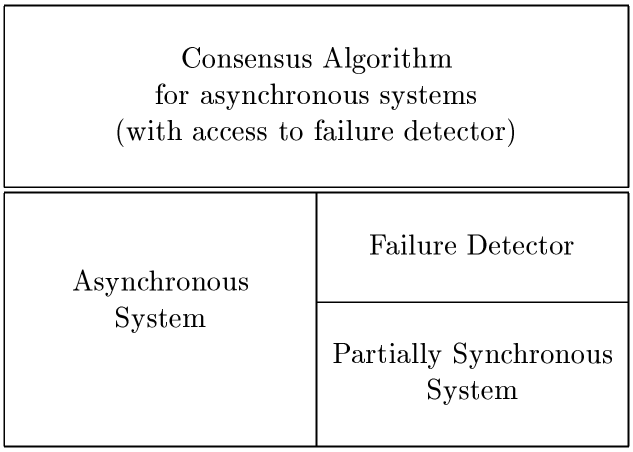

Figure 2.3: Architecture of a fault-tolerant Consensus algorithm based on a failure detector.

2.6 Related Work

2.6.1 Combining Randomization and Failure Detectors

In [AT96], Aguilera and Toueg present a Consensus algorithm that combines unreliable failure detection and randomization in asynchronous systems with crash failures. This work is motivated by the observation that the consensus algorithm based on the failure detector class $\diamond\mathcal{S}$ of [CT96] can stop making progress during "bad" periods, i.e., when the failure detector loses its accuracy. This hybrid algorithm combines advantages from both approaches:

[11] In the Consensus algorithm of Chandra and Toueg for $\diamond\mathcal{S}$, the failure detector is only used at some specific point in each round.

it guarantees deterministic termination if the failure detector is accurate, and probabilistic termination otherwise. Randomization is used as a "back-up" to ensure termination in the occasional "bad" periods when the failure detector loses its accuracy property. A similar approach is followed by Mostefaoui et al. in [MRF99].

2.6.2 Timed Asynchronous System Model and Paxos Algorithm

The *Timed Asynchronous* distributed system model [FC99] assumes that (1) all services are timed: their specification prescribes not only the outputs and state transitions that should occur in response to inputs, but also the time intervals within which a client can expect these outputs and transitions to occur, and (2) processes have access to hardware clocks with bounded drift from real-time. Hardware clocks allow one to implement application level timeouts.

This model assumes that the system alternates between long periods of stability and short periods of instability. A stable period is a period during which known bounds hold for the transmission delay of messages and the relative speeds of processes, i.e., a *synchronous* period. A progress assumption states that after an unstable period there always exists a time interval of some given minimum length in which the system will be stable. Fetzer et al. show that the timed asynchronous model is sufficiently strong to solve many practically relevant problems, such as clock synchronization, leader election, consensus, group membership, and atomic broadcast.

A similar model is considered in the Paxos algorithm [Lam96, PLL97, Lam98], which assumes a partially synchronous system model in which periods of synchrony and asynchrony alternate, and uses an *eventually perfect* failure detector mechanism to solve Consensus in periods of synchrony, provided that they are sufficiently long.

2.6.3 Other Classes of Failure Detectors

Several others classes of failure detectors have been proposed in the literature for solving problems other than Consensus, or for solving Consensus in other failure models. We briefly present them.

Failure Detectors for Quiescent Algorithms

In [ACT97a, ACT97b], Aguilera et al. study the problem of achieving reliable communication with *quiescent* algorithms (i.e., algorithms that eventually stop sending messages) in asynchronous systems with process crashes and lossy (but fair) links. They show that, among the eight classes of failure detectors defined in [CT96], the weakest one that can be used to solve this problem is $\diamond\mathcal{P}$, which cannot be implemented in purely asynchronous systems. To overcome this difficulty, they propose a new failure detector that is implementable without timeouts, called *Heartbeat*. In contrast to typical failure detectors, the Heartbeat failure detector does not output lists of suspects, but a vector of unbounded counters, one for each other process in the system. The authors show that with Heartbeat, many existing algorithms that tolerate only process crashes can be transformed into quiescent algorithms that tolerate both process crashes and message losses.

Failure Detectors for Uniform Reliable Broadcast

In [ATD99], Aguilera et al. study the problem of what failure detectors are necessary to implement *Uniform Reliable Broadcast*, a communication primitive that requires that if a process – correct or incorrect – delivers a message, then all correct processes also deliver this message [HT94]. They prove that the weakest failure detector for solving this problem is a failure detector denoted Θ. Θ outputs a set of processes that are currently *trusted* to be up,[12] such that the following completeness and accuracy properties are satisfied: (1) there is a time after which correct processes do not trust any process that crashes, and (2) if there is a correct process then, at every time, every process trusts at least one correct process. The paper also presents an algorithm that implements Θ in a system with a majority of correct processes.

Failure Detectors in the Crash/Recover Model

In [ACT98], Aguilera and Toueg study the problems of failure detection and consensus in asynchronous systems in which processes may crash and recover, and links may lose messages but are fair. They propose new failure detectors that are suitable to the crash-recovery

[12]This is just the complement of the set of processes suspected to have crashed.

model, which do not output lists of suspected processes but lists of *trusted* processes (with an associated *epoch number* for each such process), and give two consensus algorithms based on the new failure detectors, one of which requires stable storage, while the other does not require stable storage.

The same problem, i.e., Consensus in the crash/recover model, is also studied in [DFKM97, OGS97, HMR97]. These papers consider failure detectors that output lists of suspected processes, and require that *unstable* processes, i.e., processes that crash and recover indefinitely, be eventually suspected forever.

Failure Detectors in the Timed Asynchronous System Model

In [FC96], Fetzer and Cristian introduce two new classes of failure detectors, called *fail-aware*, that are (1) implementable in the timed asynchronous system model, (2) not necessarily perfect, and (3) can be used to solve the leader election problem. The main property of fail-aware failure detectors is that a process p's failure detector module knows when "too many" processes might erroneously suspect that p has crashed and notifies p of that. The authors use this property to solve the leader election problem in timed asynchronous systems.

Failure Detectors for Muteness Failures

In [DS97, DGGS99], Doudou et al. study the problem of failure detection in the context of Byzantine failures. Based on the concept of *muteness* failures, i.e., failures that cause a process to prematurely stop sending messages corresponding to the algorithm that is executed, Doudou et al. characterize a class of *Muteness* Failure Detectors. They also propose an implementation of a muteness failure detector in a partially synchronous model.

Γ-Accurate Failure Detectors, Failure Detectors with Limited Scope Accuracy, and Restricted Failure Detectors

The accuracy properties defined by Chandra and Toueg restrict the false suspicions that can be made *by all the processes in the system.* In [GS96b], Guerraoui and Schiper define "Γ-accurate" failure detectors, whose accuracy properties *only* restrict the false suspicions that can be made *by a subset* Γ *of the processes.* The paper discuss the relations between accurate and Γ-accurate failure detector classes.

The same problem is investigated by Mostefauoi and Raynal [MR99b], and Raynal and Tronel [RT99b]. This last paper extends the restricted scope also to the completeness property.

∞∞∞

Chapter 3

Impossibility of Implementing Failure Detectors with Perpetual Accuracy

3.1 Introduction

In this chapter, we show that from the eight classes of failure detectors defined by Chandra and Toueg in [CT96], no one of the four classes that satisfy *perpetual* – strong or weak – accuracy, namely $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, and $\mathcal{W}$, can be implemented deterministically in a partially synchronous system that conforms to the model M_3 [CT96] considered in this thesis[1]. Combined with the algorithms implementing failure detector classes $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, and $\diamond\mathcal{W}$ proposed in Chapter 4, this result shows that only failure detectors satisfying *eventual* accuracy can be implemented in such a partially synchronous system.

Remember that model M_3 stipulates that, in every run[2] of the system, there are upper bounds Δ and Φ on the transmission delay of messages and the relative speeds of processes respectively, but these bounds are not known *and* they hold only after some unknown but finite time (called *GST* for *Global Stabilization Time*). To strengthen our result, we will assume that communication channels are reliable. As pointed out in the previous chapter,

[1]Note that Consensus is solvable in M_3. However, this does not contradict our result, since we know that $\diamond\mathcal{W}$ (and even $\diamond\mathcal{P}$) can be implemented in M_3.

[2]The term *run* in in this chapter corresponds to the term *execution* of Chapter 2.

in this case the models M_1 of Dwork et al. [DLS88] (identical to M_3 but with the bounds holding from the beginning, i.e., $GST = 0$) and M_3 are equivalent. Hence, for simplicity we will prove the result for model M_1 (without considering GST) and then extend it to model M_3.

3.2 An Outline of the Result

From the relationship between failure detector classes described in [CT96], it is sufficient to show the impossibility result for the failure detector class $\mathcal{W}$, since $\mathcal{W}$ is the *weakest* of the four classes of failure detectors satisfying perpetual accuracy. For clarity reasons, we have preferred to proceed separately for the failure detector classes satisfying perpetual *strong* accuracy, i.e., $\mathcal{P}$ and $\mathcal{Q}$, and those satisfying perpetual *weak* accuracy, i.e., $\mathcal{S}$ and $\mathcal{W}$, respectively. In both cases, the approach followed is based on showing that no failure detector satisfying both a perpetual accuracy property and a completeness property can be implemented. In particular, we assume the existence of a failure detector satisfying a completeness property, and show that perpetual accuracy is violated.

The principle used to prove the impossibility is to consider different runs of the system – with and without crashes – such that they look identical for some correct processes up to certain time t. Hence, these processes take the same actions in all the runs up to t, in particular in what concerns the suspicion of other processes. We show that by doing this, the required perpetual accuracy is violated, and thus the failure detector does not implement any of the four classes defined in [CT96].

We first show the impossibility result for failure detector classes $\mathcal{P}$ and $\mathcal{Q}$. For that, one single incorrect suspicion of a correct process by another correct process is sufficient, since this violates the perpetual strong accuracy property. Then, we extend the result to failure detector classes $\mathcal{S}$ and $\mathcal{W}$, by showing a run of the system in which all the correct processes are erroneously suspected at least once, thus violating perpetual weak accuracy[3].

[3]Note that an algorithm implementing any given class $\mathcal{D}$ of failure detectors must satisfy the properties that characterize $\mathcal{D}$ in *all* runs.

3.3 Impossibility for Failure Detectors with Perpetual Strong Accuracy

In this section, we show the impossibility result for failure detector classes $\mathcal{P}$ and $\mathcal{Q}$. Let Σ be a partially synchronous distributed system that conforms to model M_1, made up of $n > 1$ processes, such that at least one of them is correct, i.e., at most $f < n$ of them may crash.

Theorem 1 *Let FD_Σ be a failure detector, implemented on the system Σ, that satisfies the weak completeness property. FD_Σ cannot satisfy the strong accuracy property.*

Proof: Let us consider a run R of Σ in which some process p crashes at time 0. Since FD_Σ satisfies the weak completeness property, there is a time t after which some correct process q permanently suspects p.

Let us consider now a run R' in which no process crashes, but:

- No message sent by p is received before time t. This can happen if we assume that $\Delta > t$.

- All processes except p behave exactly like in run R up to time t.

Clearly, process q cannot distinguish run R from run R' up to time t as defined in R. Hence, at time t, q will suspect p in R', as it did in R, and the strong accuracy property is not satisfied. ■

Corollary 1 *There is no protocol that implements a failure detector of class $\mathcal{Q}$ in the partially synchronous distributed system Σ.*

Corollary 2 *There is no protocol that implements a failure detector of class $\mathcal{P}$ in the partially synchronous distributed system Σ.*

Proof: Follows from Corollary 1 and the fact that $\mathcal{Q}$ and $\mathcal{P}$ are equivalent [CT96]. ■

3.4 Impossibility for Failure Detectors with Perpetual Weak Accuracy

In this section, we show the impossibility result for failure detector classes $\mathcal{S}$ and $\mathcal{W}$. We first give a more intuitive preliminary result, which assumes runs in which all processes except one crash. Then, we generalize the result for any number of crashes.

Let Σ be a partially synchronous distributed system that conforms to model M_1, made up of $n > 1$ processes.

Impossibility for $f = n - 1$

Theorem 2 *Let FD_Σ be a failure detector, implemented on the system Σ, that satisfies the strong completeness property. FD_Σ cannot satisfy the weak accuracy property.*

Proof: Let us consider a run R_i, $i = 1, \ldots, n$, of Σ in which all processes except p_i crash at time 0. Since FD_Σ satisfies the strong completeness property, there is some time t_i at which p_i suspects all other processes. Let us define $t = \max_i\{t_i\}$.

Let us consider now a run R in which no process crashes, but:

- No message sent is received before time t. This can happen if we assume that $\Delta > t$.
- Each process p_i, $i = 1, \ldots, n$, behaves exactly like in run R_i up to time t.

Clearly, a process p_i, $i = 1, \ldots, n$, cannot distinguish run R from run R_i up to time t. Hence, at time $t_i \leq t$ it will suspect the rest of processes in R, as it did in R_i. Since this is true for every process $i = 1, \ldots, n$ in the system, in run R all correct processes are suspected at some time by the rest of correct processes, and the weak accuracy property is not satisfied. ■

Corollary 3 *There is no protocol that implements a failure detector of class $\mathcal{S}$ in the partially synchronous distributed system Σ.*

Corollary 4 *There is no protocol that implements a failure detector of class $\mathcal{W}$ in the partially synchronous distributed system Σ.*

Proof: Follows from Corollary 3 and the fact that $\mathcal{S}$ and $\mathcal{W}$ are equivalent [CT96]. ■

Impossibility for $f < n$

Theorem 3 *Let FD_Σ be a failure detector, implemented on the system Σ, that satisfies the strong completeness property. FD_Σ cannot satisfy the weak accuracy property.*

Proof: Let us consider a run R_1 of Σ in which only process p_1 crashes, doing it at time 0. Since FD_Σ satisfies the strong completeness property, there is some time t_1 at which all other processes permanently suspect p_1 in R_1.

Let us consider a run R_2 of Σ in which only process p_2 crashes, doing it at time t_1 as defined in R_1, and no message sent by p_1 is received before t_1 (this can happen if we assume that $\Delta > t_1$). Also in R_2, all processes except p_1 behave exactly like in run R_1 up to time t_1. Clearly, all correct processes except p_1 cannot distinguish run R_2 from run R_1 up to time t_1. Hence, at time t_1 they will suspect p_1 in R_2, as they did in R_1. Finally, since FD_Σ satisfies the strong completeness property, there is some time $t_2 \geq t_1$ at which all other processes permanently suspect p_2 in R_2.

By a similar reasoning, let us consider a run R_i, $i = 2, \ldots, n$ of Σ in which only process p_i crashes, doing it at time t_{i-1} as defined in R_{i-1}, and for each process p_k, $k = 1, \ldots, i-1$, no message sent by p_k after t_{k-1} is received before t_k, with $t_0 = 0$ (this can happen if we assume that $\forall k : 1, \ldots, i-1 : \Delta > max(t_k - t_{k-1})$, with $t_0 = 0$). Also in R_i, for each process p_k, $k = 1, \ldots, i-1$, all processes except p_k behave exactly like in run R_k up to time t_k. Clearly, for each process p_k, $k = 1, \ldots, i-1$, all correct processes except p_k cannot distinguish run R_i from run R_k up to time t_k. Hence, at time t_k they will suspect p_k in R_i, as they did in R_k. Finally, since FD_Σ satisfies the strong completeness property, there is some time $t_i \geq t_{i-1}$ at which all other processes permanently suspect p_i in R_i.

Let us now consider a run R of Σ in which no process crashes, but:

- No message sent by p_n after time t_{n-1} as defined in R_{n-1} is received before time t_n as defined in R_n. This can happen if we assume that $\forall i : 1, \ldots, n : \Delta > max(t_i - t_{i-1})$, with $t_0 = 0$.
- For each process p_i, $i = 1, \ldots, n$, all processes except p_i behave exactly like in run R_i up to time t_i.

Clearly, for each process p_i, $i = 1, \ldots, n$, all processes except p_i cannot distinguish run R from run R_i up to time t_i. Hence, at time t_i they will suspect p_i in R, as they did in R_i. Since this is true for every process p_i, $i = 1, \ldots, n$ in the system, in run R all correct processes are suspected at some time by the rest of correct processes, and the weak accuracy property is not satisfied. ∎

Corollary 5 *There is no protocol that implements a failure detector of class $\mathcal{S}$ in the partially synchronous distributed system Σ.*

Corollary 6 *There is no protocol that implements a failure detector of class $\mathcal{W}$ in the partially synchronous distributed system Σ.*

Proof: Follows from Corollary 5 and the fact that $\mathcal{S}$ and $\mathcal{W}$ are equivalent [CT96]. ∎

∞∞∞

Chapter 4

Implementation of Failure Detectors with Eventual Accuracy

4.1 Introduction

In this chapter, we propose efficient algorithms implementing the four classes of failure detectors defined in [CT96] that satisfy *eventual* accuracy, namely:

- *Eventually Perfect ($\diamond\mathcal{P}$)*, which satisfies strong completeness and eventual strong accuracy
- *Eventually Quasi-Perfect ($\diamond\mathcal{Q}$)*, which satisfies weak completeness and eventual strong accuracy
- *Eventually Strong ($\diamond\mathcal{S}$)*, which satisfies strong completeness and eventual weak accuracy
- *Eventually Weak ($\diamond\mathcal{W}$)*, which satisfies weak completeness and eventual weak accuracy

We first present a reference algorithm that implements the *Eventually Perfect* failure detector class, $\diamond\mathcal{P}$. This algorithm has been proposed by Chandra and Toueg in [CT96], and it is based on an idea of Dwork et al. [DLS88]. Next, we propose more efficient algorithms that implement $\diamond\mathcal{W}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, and $\diamond\mathcal{P}$ respectively. These algorithms are based on a ring arrangement of processes, and are designed in a modular way, i.e., by extending a basic algorithm. We also propose an optimal algorithm implementing the *Eventually Strong* failure detector class, $\diamond\mathcal{S}$, which is the weakest failure detector for solving Consensus[1]. Finally, we present performance evaluation results of all the previous algorithms obtained by discrete event simulation.

4.2 System Model

The system model considered in this chapter is the following. We consider a distributed system consisting of a finite set Π of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving messages. Every pair of processes is assumed to be connected by a quasi-reliable communication channel.

Processes can fail by *crashing*, that is, by prematurely halting. Crashes are permanent, i.e., crashed processes do not recover. In every execution of the system we identify two complementary subsets of Π: the subset of processes that do not fail, denoted *correct*, and the subset of processes that do fail, denoted *crashed*. We use $\mathcal{C}$ to denote the number of correct processes in the system, which we assume is at least one, i.e., $\mathcal{C} = |correct| > 0$.

We consider a partially synchronous system conforming to the model M_3 [CT96]. This model stipulates that, in every execution, there are upper bounds Δ and Φ on message transmission time and processor relative speeds respectively, but these bounds are not known *and* they hold only after some unknown – but finite – time (called GST for *Global Stabilization Time*). We shall use T_s to denote the ending instant of this stabilization interval. We will also denote by Δ_{msg} the maximum time, after GST, between the sending of a message and the delivery and processing by its destination process (assuming that both the sender and the destination have not crashed). Clearly, Δ_{msg} depends on the existing bounds on processor relative speeds and on message transmission times. Note that the exact value of Δ_{msg} exists, but it is unknown.

We consider that a failure detector is composed by a set of n failure detection modules, each one attached to a different process in the system. These modules maintain a local list of the processes they suspect to have crashed, and cooperate, i.e., exchange messages, to satisfy the required properties of the failure detector. We assume that a process interacts only with its local failure detection module in order to get the current list of suspected processes.

[1] Actually, the *Eventually Weak* failure detector class, $\diamond\mathcal{W}$, is presented as the weakest one for solving Consensus [CHT96]. However, Chandra and Toueg have shown in [CT96] that $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent.

An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in *steps* of the algorithm. In each step, a process may (1) send a message to another process, (2) receive a message sent by another process, (3) perform some local computation, e.g., query its local failure detector module, and (4) fail by crashing.

In this chapter, we only describe the behavior of the failure detection modules in order to implement a failure detector, but not the behavior of the processes they are attached to. For this reason, in the rest of the chapter we will mostly use the term *process* instead of *failure detection module*. It will be clear from the context if we are referring to the failure detection module or the process attached to it. We consider that a process cannot crash independently of its attached failure detection module.

4.2.1 Monitoring Models

Any algorithm implementing a failure detector requires that some processes detect whether other processes have crashed, and take proper action if so (usually suspect them). Clearly, the common way a process can show it has not crashed in a distributed system is by sending messages to the processes monitoring it. There are mainly two monitoring models: the *push* model and the *pull* model [Gro98].

In the push model (also called *heartbeat* model), monitored processes are active. They periodically send I-AM-ALIVE messages to inform other processes that they are still alive. If a monitoring process p does not receive the I-AM-ALIVE message from a monitored process q within an expected timeout period[2], then p suspects that q has crashed.

In the pull model (also called *polling* or *interrogation* model), monitored processes are passive. The processes in charge of monitoring periodically send ARE-YOU-ALIVE? messages – liveness requests – to monitored processes. If a monitored process replies on time with a corresponding I-AM-ALIVE message, it means that it is alive. Otherwise, a timeout triggers a suspicion in the monitoring process.

The algorithms presented in Sections 4.3 and 4.5 are based on the heartbeat monitoring model (with different communication patterns), while the ring-based algorithms presented in Section 4.4 are based on the polling monitoring model.

[2]All the failure detectors presented in this chapter are based on a timeout mechanism, i.e., they detect the crash of a process after the expiration of a certain timeout.

4.3 A Reference Implementation of $\Diamond\mathcal{P}$

In this section, we present a reference algorithm implementing an *Eventually Perfect* ($\Diamond\mathcal{P}$) failure detector. This algorithm has been proposed by Chandra and Toueg in [CT96], and it is based on an idea of Dwork et al. [DLS88]

The algorithm works as follows. Every process q periodically sends a "q-is-alive" message to all processes. If a process p times-out on some process q, it adds q to its list of suspects. If p later receives a "q-is-alive" message, p recognizes that it made a mistake by prematurely timing out on q: p removes q from its list of suspects, and increases the length of its timeout period for q in an attempt to prevent a similar mistake in the future. If there are n processes in the system and $\mathcal{C}$ of them do not crash, then at least $n\mathcal{C}$ messages are periodically exchanged with this algorithm. Figure 4.1 shows the algorithm in detail.

Every process p executes the following:

$output_p \leftarrow \emptyset$
for all $q \in \Pi$ {$\Delta_p(q)$ *denotes the duration of p's time-out interval for q*}
  $\Delta_p(q) \leftarrow$ default time-out interval

cobegin

|| *Task 1:* **repeat periodically**
  send "p-is-alive" to all

|| *Task 2:* **repeat periodically**
  for all $q \in \Pi$
    if $q \notin output_p$ and
      p did not receive "q-is-alive" during the last $\Delta_p(q)$ ticks of p's clock
      $output_p \leftarrow output_p \cup \{q\}$ {*p times-out on q: it now suspects q has crashed*}

|| *Task 3:* **when** receive "q-is-alive" for some q
  if $q \in output_p$ {*p knows that it prematurely timed-out on q*}
    $output_p \leftarrow output_p - \{q\}$ {*1. p repents on q, and*}
    $\Delta_p(q) \leftarrow \Delta_p(q) + 1$ {*2. p increases its time-out period for q*}

coend

Figure 4.1: A timeout based algorithm implementing $\Diamond\mathcal{P}$ in models of partial synchrony.

4.4 Efficient Implementations of $\diamond\mathcal{W}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, and $\diamond\mathcal{P}$

In this section, we present a family of algorithms that implement the four classes of unreliable failure detectors satisfying eventual accuracy, namely $\diamond\mathcal{W}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, and $\diamond\mathcal{P}$. The algorithms are based on a logical ring arrangement of the processes, which defines the monitoring and failure information propagation pattern. As a consequence, the resulting algorithms periodically exchange at most a linear number of messages.

The algorithms have been designed, and are presented, in a modular way. First, we present a basic algorithm providing *only* weak completeness (i.e., it does not provide any accuracy). Next, we present two extensions to this basic algorithm that provide the two eventual accuracy properties respectively, and a third extension that transforms weak completeness into strong completeness, while preserving accuracy. Hence, the basic algorithm combined with the extensions implement the four classes of failure detectors.

4.4.1 Overview of the Algorithms

In the algorithms presented in this section we consider the processes $p_1, \ldots, p_n$ arranged in a logical ring. This arrangement is known by all the processes. Without loss of generality, process p_i is followed in the ring by process $p_{(i \bmod n)+1}$. In general, we use $succ(p)$ to denote the process that follows process p in the ring, and $pred(p)$ to denote the process that precedes process p in the ring. Finally, we use $corr\_succ(p)$ and $corr\_pred(p)$ to denote the closest correct (i.e., belonging to the subset *correct*) successor and predecessor of p in the ring, respectively.

We denote by L_p (G_p in Section 4.4.5) the list of suspected processes of the failure detection module attached to process p. Clearly, the contents of the list L_p (G_p) can be different at different times. We use $L_p(t)$ ($G_p(t)$ in Section 4.4.5) to denote the contents of L_p (G_p) at time t.

The algorithms use a polling monitoring model. To monitor process q, a process p sends an ARE-YOU-ALIVE? message to q and waits for an I-AM-ALIVE message from it. As soon as q receives the ARE-YOU-ALIVE? message, it sends the I-AM-ALIVE message to p. We will denote by $\Delta_{rtt} = 2\Delta_{msg}$ the maximum monitoring round-trip time after stabilization, i.e., the maximum time, after T_s, elapsed between the sending of an ARE-YOU-ALIVE? message to a correct process, and the reception and processing of the corresponding I-AM-ALIVE reply message.

Since a monitoring process p does not know Δ_{rtt}, it has to use an estimated value (timeout) that tells how much time it has to wait for the reply from the monitored process q. This time value is denoted by $\Delta_{p,q}$. Then, if after $\Delta_{p,q}$ time p did not receive the reply from q, it suspects that q has crashed. We need to allow these time values to vary over time in the algorithms. We use $\Delta_{p,q}(t)$ to denote the value of $\Delta_{p,q}$ at time t.

In all these algorithms, each correct process monitors only one other process in a cyclic fashion. The monitoring process performs this task by repeatedly polling the monitored process. Each polling involves only two messages exchanged between the monitoring and monitored processes. If the pollings were done periodically, a total of no more than $2n$ messages would be periodically exchanged. Eventually, this amount becomes at most $2\mathcal{C}$, which is a significant improvement over the at least $n\mathcal{C}$ messages of Chandra and Toueg's algorithm.

4.4.2 A Basic Algorithm that Provides Weak Completeness

We present here an algorithm that will be used as a framework for all the failure detector implementations presented in this section. This first algorithm satisfies the weak completeness property. In the following sections, we will extend the algorithm to satisfy also eventual weak accuracy, eventual strong accuracy, and strong completeness. This algorithm is presented here for the sake of clarity but is not very useful by itself, since it does not satisfy any of the accuracy properties.

The algorithm executes as follows: initially, every process starts monitoring its successor in the ring. If a process p does not receive the reply from the process q it is monitoring, then p suspects that q has crashed, and starts monitoring the successor of q in the ring. This monitoring scheme is repeated, so that p always suspects all processes in the ring between itself and the process it is monitoring (not included). If, later on, p receives a message from a suspected process q while it is monitoring another process r, then p stops suspecting q and all the processes between q and r in the ring, and starts monitoring q again.

Figure 4.2 presents the algorithm in detail. Each process p has a variable $target_p$ which holds the process being monitored by p at a given time. As we said above, all processes between p and $target_p$ in the ring (and only them) are suspected by p, and these are the only processes included in the list L_p of suspected processes of p. (Initially, no process is suspected, i.e., $\forall p : L_p(0) = \emptyset$.)

Every process p executes:

$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin

|| Task 1:
loop
send ARE-YOU-ALIVE? to $target_p$
$t_{out} \leftarrow \Delta_{p,target_p}$
$received \leftarrow false$
delay t_{out}
if not $received$ **then**
$L_p \leftarrow L_p \cup \{target_p\}$
$target_p \leftarrow succ(target_p)$
end if
end loop

|| Task 2:
loop
receive message m from a process q
if $q = target_p$ **then**
$received \leftarrow true$
else if $q \in L_p$ **then**
$L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
$target_p \leftarrow q$
$received \leftarrow true$
end if
if $m =$ ARE-YOU-ALIVE? **then**
send I-AM-ALIVE to q
end if
end loop

coend

Figure 4.2: Algorithm that provides weak completeness.

We now show that weak completeness holds with this algorithm. Given an incorrect process p, the following theorem states that it will be permanently suspected by $corr\_pred(p)$ (the first correct process preceding p in the ring).

Theorem 4 *$\exists t_0 : \forall p \in crashed$, p has failed at time t_0 and $\forall t \geq t_0, p \in L_{corr\_pred(p)}(t)$.*

Proof: Let p be a process that crashes, doing it at time $Tcrash_p$. We claim that p will be permanently included in $L_{corr\_pred(p)}$. The proof uses strong induction on the distance from $corr\_pred(p)$ to p. Let first consider that such distance is 1, i.e., $corr\_pred(p) = pred(p)$. Before p fails, $corr\_pred(p)$ and p exchange ARE-YOU-ALIVE? and I-AM-ALIVE messages (see Figure 4.2). Eventually p crashes, and there is an ARE-YOU-ALIVE? message sent by $corr\_pred(p)$ that reaches p after $Tcrash_p$. Since p has already crashed by then, it will never reply to that message. If such a message was sent at time t', then $\Delta_{corr\_pred(p),p}(t')$ time later, $corr\_pred(p)$ will include p in $L_{corr\_pred(p)}$. Since no message will ever be received by $corr\_pred(p)$ from p after that, it will never be removed from $L_{corr\_pred(p)}$.

We will now prove that if the claim holds for any distance $1 \leq d \leq i-1$, it also holds for distance i. Let us assume the distance from $corr\_pred(p)$ to p be $i > 1$. Then, for any process $q \in \{succ(corr\_pred(p)), \ldots, pred(p)\}$, it can be easily seen that $corr\_pred(q) = corr\_pred(p)$ and the distance d from $corr\_pred(p)$ to q verifies $1 \leq d \leq i-1$. Hence, from the induction hypothesis, all processes in $\{succ(corr\_pred(p)), \ldots, pred(p)\}$ will eventually be permanently in $L_{corr\_pred(p)}$. After that, they will never be monitored again by $corr\_pred(p)$. The situation then is similar to the distance-1 case considered above and, by a similar argument, p will eventually be permanently included in $L_{corr\_pred(p)}$. ■

Corollary 7 *The algorithm of Figure 4.2 provides weak completeness.*

4.4.3 Extending the Basic Algorithm to Provide Eventual Weak Accuracy

The algorithm presented in the previous section does not satisfy any of the eventual accuracy properties defined in [CT96]. It does not prevent the erroneous suspicion of any correct process, and these incorrect suspicions, although not permanent (if the suspected process is correct, the reply message will eventually be received), can happen infinitely often. This is due to the fact that the message delivery time could be greater than the fixed *default timeout* (see Figure 4.2). In order to provide some useful accuracy, the timeout values must be augmented when processes are aware of having erroneously suspected a correct process. In this section, we present an extension to the basic algorithm of Figure 4.2, based on augmenting the timeout values, which satisfies the eventual weak accuracy property.

Eventual weak accuracy requires that, eventually, some correct process is never suspected by any correct process. In order to provide it, it is enough that this is satisfied for only one correct process. We propose an extension to the basic algorithm that guarantees the existence of such a process, which we denote *leader*. Clearly, if we knew beforehand a correct process, eventual weak accuracy could be obtained by making all processes augment their timeout value with respect to this process each time they suspect it. This correct process would be *leader*. But since we cannot know in advance the correctness of any process, we need to devise another way to eventually have a correct and not-suspected process.

In the extension of the algorithm of Figure 4.2 that we propose, processes behave as follows. Initially, every process will consider a pre-agreed process (e.g. p_1) as an *initial candidate* to be *leader*. When a process that monitors this candidate suspects it, it considers its successor in the ring as new candidate and monitors it. This scheme is repeated every time the current candidate is suspected. (Note that a process not monitoring a candidate cannot suspect it.) If a process p stops suspecting a process q, previously considered as candidate, then p will augment its timeout value $\Delta_{p,q}$ [3]. If the previously suspected process q was not considered as candidate, then p will not change $\Delta_{p,q}$. This way, *leader* will be the first correct process in the ring starting from the *initial candidate* (inclusive). All processes monitoring it will eventually stop suspecting it, and processes that do not monitor it will never suspect it. This gives us the eventually weak accuracy property. Figure 4.3 presents the extended algorithm in detail.

We now show that eventual weak accuracy holds with this algorithm, i.e., eventually some correct process is never suspected by any correct process.

Lemma 1 *After T_s, any correct process p will suspect leader for no more than Δ_{rtt} time, each time it does.*

Proof: Remember that, after T_s, Δ_{rtt} is a bound on the monitoring round-trip time. A correct process p suspects *leader* after sending an ARE-YOU-ALIVE? message to it at time t and not receiving an I-AM-ALIVE reply message in $\Delta_{p,leader}(t)$ time. Since, by definition, *leader* is a correct process, the I-AM-ALIVE message will arrive at most at time $t + \Delta_{rtt}$ ($T_s + \Delta_{rtt}$ if $t < T_s$). At this moment *leader* is removed from L_p, the list of suspected processes of p. ■

Lemma 2 *Any correct process p will suspect leader a finite number of times.*

Proof: The proof is by contradiction. Let p be some correct process. Let us assume that p suspects *leader* an infinite number of times. From the algorithm, each time p suspects *leader*, the value of $\Delta_{p,leader}$ is incremented by one. Since T_s and Δ_{rtt} are finite (although unknown),

[3] For simplicity of the algorithm, instead of increasing timeouts when we stop suspecting, we increase timeouts as soon as we suspect a candidate.

```
Every process p executes:

initial_cand_p ← pre-agreed process, e.g., p_1
target_p ← succ(p)
L_p ← ∅
∀q ∈ Π : Δ_{p,q} ← default timeout

cobegin

|| Task 1:
  loop
    send ARE-YOU-ALIVE? to target_p
    t_out ← Δ_{p,target_p}
    received ← false
    delay t_out
    if not received then
      if initial_cand_p ∈ {succ(p), ..., target_p} then
        Δ_{p,target_p} ← Δ_{p,target_p} + 1
      end if
      L_p ← L_p ∪ {target_p}
      target_p ← succ(target_p)
    end if
  end loop

|| Task 2:

    { Same as in Figure 4.2 }

coend
```

Figure 4.3: Extension to the algorithm of Figure 4.2 to provide eventual weak accuracy.

after suspecting *leader* a finite number of times, $\Delta_{p,leader}$ will be greater than Δ_{rtt}. After this moment, p never suspects *leader* anymore, and thus we reach a contradiction. ■

Theorem 5 $\exists t_1 : \forall p \in correct, \forall t > t_1, leader \notin L_p(t)$

Proof: Let t_1^p be the instant at which a correct process p stops suspecting *leader* for the last time. (If p never suspects *leader*, $t_1^p = 0$.) Such an instant exists from Lemma 1 and Lemma 2. Then, after instant $t_1 = \max_{p \in correct} \{t_1^p\}$ no correct process p has *leader* in its list L_p. ■

Corollary 8 *The algorithm of Figure 4.3 provides eventual weak accuracy.*

Observation 1 *The only difference between the algorithm of Figure 4.3 and the algorithm of Figure 4.2 is that in the former the values of $\Delta_{p,q}$ can change. Clearly, this does not affect the proof of Theorem 4. Hence, Corollary 7 also applies to this algorithm.*

Corollary 9 *The algorithm of Figure 4.3 implements a failure detector of class $\Diamond\mathcal{W}$.*

Proof: Follows from Corollary 8, Observation 1, and Corollary 7. ■

4.4.4 Extending the Basic Algorithm to Provide Eventual Strong Accuracy

Eventual strong accuracy requires that, eventually, no correct process is ever suspected by any correct process. In this section, we propose another extension to the basic algorithm of Figure 4.2 which satisfies this property. Broadly, the extension consists in each process augmenting its timeout values with respect to all processes it incorrectly suspects. This way every process will augment the timeout value with respect to its closest correct successor in the ring, and will thus eventually stop suspecting it (and hence, any other correct process). This gives us the eventually strong accuracy property. Figure 4.4 presents the extended algorithm in detail.

We now show that eventual strong accuracy holds with the algorithm in Figure 4.4. We start with two lemmas, whose proofs are very similar to those of Lemma 1 and Lemma 2, respectively, and are omitted.

Lemma 3 *After T_s, any correct process p will suspect $corr\_succ(p)$ for no more than Δ_{rtt} time, each time it does.*

Lemma 4 *Any correct process p will suspect $corr\_succ(p)$ a finite number of times.*

Theorem 6 $\exists t_2 : \forall p \in correct, \forall q \in correct, \forall t > t_2, q \notin L_p(t)$

Proof: Let t_2^p be the instant at which a correct process p stops suspecting $corr\_succ(p)$ for the last time. (If p never suspects $corr\_succ(p)$, $t_2^p = 0$.) Such an instant exists from Lemma 3 and Lemma 4. Then, after instant $t_2 = \max_{p \in correct}\{t_2^p\}$ no correct process p has $corr\_succ(p)$ in its list L_p. Then, after t_2, each correct process p only suspects processes in $succ(p), \ldots, pred(corr\_succ(p))$, which are not correct by the definition of $corr\_succ(p)$.

```
Every process p executes:

target_p ← succ(p)
L_p ← ∅
∀q ∈ Π : Δ_{p,q} ← default timeout

cobegin

|| Task 1:                                      || Task 2:
  loop
    send ARE-YOU-ALIVE? to target_p                { Same as in Figure 4.2 }
    t_out ← Δ_{p,target_p}
    received ← false
    delay t_out
    if not received then
      Δ_{p,target_p} ← Δ_{p,target_p} + 1
      L_p ← L_p ∪ {target_p}
      target_p ← succ(target_p)
    end if
  end loop

coend
```

Figure 4.4: Extension to the algorithm of Figure 4.2 to provide eventual strong accuracy.

Therefore, no correct process q is in L_p after t_2. ■

Corollary 10 *The algorithm of Figure 4.4 provides eventual strong accuracy.*

Note that Observation 1 still applies to the algorithm of Figure 4.4. Hence, the following corollary, that follows from Corollary 10, Observation 1, and Corollary 7.

Corollary 11 *The algorithm of Figure 4.4 implements a failure detector of class $\diamond\mathcal{Q}$.*

4.4.5 Extending the Previous Algorithms to Provide Strong Completeness

In this section we present an extension to the previous algorithms to provide strong completeness, while preserving accuracy. By combining this extension with the algorithms that implement failure detectors of classes $\diamond\mathcal{W}$ and $\diamond\mathcal{Q}$ of Figure 4.3 and Figure 4.4, we obtain implementations of failure detectors of classes $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$, respectively.

Strong completeness requires that, eventually, every process that crashes is permanently suspected by every correct process. In [CT96], Chandra and Toueg presented a distributed algorithm that transforms weak completeness into strong completeness. Informally, in their algorithm every process periodically *broadcasts* (sends to every other process) its *local* list of suspected processes. Upon reception of these lists, each process builds a *global* list of suspected processes, which provides strong completeness. Clearly, in this algorithm each correct process periodically sends n messages, with the total number of messages exchanged being at least $n\mathcal{C}$ (with $\mathcal{C}$ being the number of correct processes).

We propose an extension that follows a similar approach. Besides its local list L_p of suspected processes, each process p has a global list G_p of suspected processes. While L_p only holds the suspected processes between p and the process p is monitoring ($target_p$), G_p holds all the processes that are being suspected in the system. Now, the global lists are the ones providing strong completeness.

In order to correctly build the global lists, processes need to propagate their local lists. However, instead of periodically broadcasting its local list, every process will only send its global list (which contains the local list) to the process it is monitoring. This process, upon reception of that list, updates its global list and further propagates it. Note that, since we use the ring arrangement of processes, each process at most sends and receives one message periodically, and the total number of messages exchanged in a period is $O(n)$ in the worst case, which eventually becomes $O(\mathcal{C})$. Furthermore, instead of using specific messages to send the global lists, we can piggyback the global lists in the ARE-YOU-ALIVE? messages inherent to the monitoring mechanism. This way, there is no increment in message exchanges from the previous algorithms. Figure 4.5 presents the extended algorithm in detail. We now show that strong completeness holds, while accuracy is preserved, with this algorithm.

Observation 2 *The only difference between this algorithm and the previous ones is the handling of the global lists of suspected processes G_p, while the local lists L_p are handled as before. Hence, Theorem 4 and whichever corresponds of Theorem 5 and Theorem 6 are still applicable to this algorithm.*

Observation 3 $\forall p \in \Pi, \forall t, L_p(t) \subseteq G_p(t)$.

```
Every process p executes:

{if the algorithm needs it:
initial_cand_p ← pre-agreed process, e.g., p_1}
target_p ← succ(p)
L_p ← ∅
G_p ← ∅
∀q ∈ Π : Δ_{p,q} ← default timeout

cobegin

|| Task 1:
  loop
    send ARE-YOU-ALIVE?
        —with G_p— to target_p
    t_out ← Δ_{p,target_p}
    received ← false
    delay t_out
    if not received then
      {Update Δ_{p,target_p} if required}
      G_p ← G_p ∪ {target_p}
      L_p ← L_p ∪ {target_p}
      target_p ← succ(target_p)
    end if
  end loop

|| Task 2:
  loop
    receive message m from a process q
    if q = target_p then
      received ← true
    else if q ∈ L_p then
      L_p ← L_p − {q, ..., pred(target_p)}
      G_p ← G_p − {q}
      target_p ← q
      received ← true
    end if
    if m = ARE-YOU-ALIVE? —with G_q— then
      send I-AM-ALIVE to q
      G_p ← G_q ∪ L_p − {p, q}
    end if
  end loop

coend
```

Figure 4.5: Extension to the previous algorithms to provide strong completeness.

Observation 4 *$\forall p \in correct, \forall t$, p will eventually receive* ARE-YOU-ALIVE? *messages after time t.*

Lemma 5 *$\exists t_3 : \forall q \in crashed, \forall p \in correct, \forall t \geq t_3, q \in G_p(t)$.*

Proof: Let us assume we are at least at instant t_0 as defined in Theorem 4. We know that at this instant any process $q \in crashed$ has already failed and has been permanently included in $L_{corr\_pred(q)}$.

Let us assume now that we have a process $q \in crashed$ and a process $p \in correct$. We claim that q will eventually be permanently included in G_p. We use strong induction on the number of correct processes in the set $\{corr\_pred(q), \ldots, p\}$. For the base case we assume there is only one correct process in the set, i.e., $p = corr\_pred(q)$. Hence, from Theorem 4, q is permanently in L_p and, from Observation 3, q will be permanently in G_p in this case.

We will now prove that, if the claim holds for any number $1 \leq c \leq i-1$ of correct processes in the set $\{corr\_pred(q), \ldots, p\}$, it also holds when the number of correct processes in the set is i. To do so, we show first that there is a time t' after which p receives ARE-YOU-ALIVE? messages and all of them carry global lists containing q. From that, it is immediate to see in the algorithm that, after receiving the first such ARE-YOU-ALIVE? message, q will be permanently included in G_p. Let us assume the number of correct processes in the set $\{corr\_pred(q), \ldots, p\}$ be $i > 1$. By induction hypothesis, there is a time t'' at which any correct process $r \in \{corr\_pred(q), \ldots, corr\_pred(p)\}$ permanently contains q in its global list G_r. Also, there is a time $t' = \max(t'', T_s) + \Delta_{msg}$ at which all the ARE-YOU-ALIVE? messages sent to p before t'' have been received. From Observation 4, process p will receive new ARE-YOU-ALIVE? messages after t'. Let be an ARE-YOU-ALIVE? message received by p from a process s at a time $t > t'$. There are two cases to consider:

- $s \in \{corr\_pred(q), \ldots, corr\_pred(p)\}$. In this case, from the induction hypothesis and the definition of t', we know that the global list G_s carried by the ARE-YOU-ALIVE? message contains q.

- $s \in \{p, \ldots, corr\_pred(corr\_pred(q))\}$. In this case, it can be seen from the algorithm that if p receives an ARE-YOU-ALIVE? message from s, then necessarily, at the time of sending the message, $p = target_s$ and L_s contained q. Therefore, from Observation 3, the G_s carried by the ARE-YOU-ALIVE? message contains q.

■

The following lemma states that the algorithm of Figure 4.5 preserves eventual accuracy.

Lemma 6 *Let p be any correct process. If there is a time after which no correct process q contains p in L_q, then there is a time after which no correct process q contains p in G_q.*

Proof: Let us assume we are at least at instant t_0 as defined in Theorem 4. We know that at this instant any process in *crashed* has already failed. Let p be a correct process and $t''' \geq t_0$ be an instant such that $\forall t \geq t''', \forall q \in correct, p \notin L_q$.

Let us assume now that we have a process $q \in correct$. We claim that there is a time after which p is never in G_q. We use strong induction on the number of correct processes in the set $\{p, \ldots, q\}$. For the base case, we assume there is only one correct process in the set, i.e., $p = q$. It is easy to observe from the algorithm that p will never include itself in G_p.

We will now prove that, if the claim holds for any number $1 \leq c \leq i-1$ of correct processes in the set $\{p, \ldots, q\}$, it also holds when the number of correct processes in the set is i. To do so, we show first that there is a time t' after which q receives ARE-YOU-ALIVE? messages and all of them carry global lists not containing p. From that, it is immediate to see in the algorithm that, after receiving the first such ARE-YOU-ALIVE? message, p will be removed (if needed) and never included again in G_q. Let us assume the number of correct processes in the set $\{p, \ldots, q\}$ be $i > 1$. By induction hypothesis, there is a time t'' after which any correct process $r \in \{p, \ldots, corr\_pred(q)\}$ does not contain p in its global list G_r. Also, there is a time $t' = \max(t'', T_s) + \Delta_{msg}$ at which all the ARE-YOU-ALIVE? messages sent to q before t'' have been received. From Observation 4, process q will receive new ARE-YOU-ALIVE? messages after t'. Let be an ARE-YOU-ALIVE? message received by q from a process s at a time $t > t'$. There are two cases to consider:

- $s \in \{p, \ldots, corr\_pred(q)\}$. In this case, from the induction hypothesis and the definition of t', we know that the global list G_s carried by the ARE-YOU-ALIVE? message does not contain p.

- $s \in \{q, \ldots, corr\_pred(p)\}$. This case cannot happen, because it would imply that p is in the local list L_s.

■

Combining both lemmas it is immediate to derive the following theorem.

Theorem 7 *The algorithm of Figure 4.5 provides strong completeness while preserving accuracy.*

Corollary 12 *The algorithm of Figure 4.5, combined with the algorithm of Figure 4.3, implements a failure detector of class $\Diamond\mathcal{S}$.*

Corollary 13 *The algorithm of Figure 4.5, combined with the algorithm of Figure 4.4, implements a failure detector of class $\Diamond\mathcal{P}$.*

4.4.6 Performance Analysis

In this section, we will evaluate the performance of the presented ring-based algorithms in terms of the number and size of the exchanged messages. Observe that failure detection is an on-going activity that inherently requires an infinite number of messages. Furthermore, the pattern of message exchange between processes can vary over time (and need not be periodic), and different algorithms can have completely different patterns. For these reasons, we have to make some assumptions in order to use the number of messages as a meaningful performance measure. We will first assume that the algorithms execute in a periodic[4] fashion, so that we can count the number of messages exchanged in a period. Secondly, to be able to compare the number of messages exchanged by different algorithms, we must assume that their respective periods have the same length.

Under the above assumptions, in the ring-based algorithms each correct process periodically polls only one other process. Each polling involves two messages. Thus, a total of no more than $2n$ messages would be periodically exchanged. Eventually, this amount becomes $2\mathcal{C}$, since there will be only $\mathcal{C}$ correct processes remaining in the system. This compares favorably with Chandra and Toueg's algorithm, which requires a periodic exchange of at least $n\mathcal{C}$ messages.

Concerning the size of the messages, the ring-based algorithms implementing failure detectors with weak completeness, i.e., $\diamond\mathcal{W}$ and $\diamond\mathcal{Q}$, require messages of $\Theta(\log n)$ bits (to identify the sender). On the other hand, the ring-based algorithms implementing failure detectors with strong completeness, i.e., $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$, require messages of $\Theta(n)$ bits, since we can code the global list G_p of suspected processes in n bits (one bit per process).

Chandra and Toueg's algorithm, which implements $\diamond\mathcal{P}$, requires messages of $\Theta(\log n)$ bits. This size is smaller than the size needed by the ring-based algorithm implementing $\diamond\mathcal{P}$. However, the total amount of information periodically exchanged in the ring-based algorithms is $\Theta(n^2)$ bits, while in Chandra-Toueg's it is $\Theta(n^2 \log n)$ bits. Furthermore, each message that is sent involves a fixed overhead. In this sense, the ring-based algorithms present an edge, since it involves less messages.

[4]For algorithms based on the heartbeat model, we assume that all processes send I-AM-ALIVE messages with the same period. For algorithms based on the polling model, we assume that there are no incorrect suspicions, and that all the timeout values are identical. Thus, the timeout value can be viewed as the periodicity of the algorithm.

A drawback of the ring-based algorithms satisfying strong completeness is that the use of the ring to propagate the information about crashes in order to obtain strong completeness delays the detection of faulty processes (relative to the use of direct message exchanges between all pairs of processes). As it will be shown in the results obtained by simulation, this time increases linearly with the number of processes in the system, while in the case of the algorithm of Chandra and Toueg of Section 4.3 remains practically constant. Clearly, there is a trade-off between the number of *extra* messages sent and the latency of failure information propagation. For example, we could propagate the list of suspects following the two directions of the ring. This would reduce the latency approximately to the half, at the price of increasing the number of messages.

4.5 An Optimal Implementation of $\diamond\mathcal{S}$

In this section, we present a new algorithm implementing $\diamond\mathcal{S}$, the weakest failure detector class for solving Consensus. This algorithm guarantees that eventually all the correct processes agree on a common correct process. This property allows us to provide the accuracy required by $\diamond\mathcal{S}$. Then, by suspecting all the other processes, we trivially obtain the completeness required by $\diamond\mathcal{S}$.

We show, then, that this algorithm performs better than existing previous algorithms implementing $\diamond\mathcal{S}$ in terms of the number of messages and the total amount of information periodically sent. In particular, previous algorithms require to periodically exchange at least a quadratic amount of information, while this algorithm only requires $O(n \log n)$ (where n is the number of processes).

We also propose a new measure to evaluate the efficiency of algorithms implementing failure detectors, called the *eventual monitoring degree*, which does not rely on a periodic behavior and expresses better the degree of processing required by the algorithms. We show that the runs of the presented algorithm have optimal eventual monitoring degree.

4.5.1 Overview of the Algorithm

We consider that processes are ordered. Without loss of generality, process p_i is preceded by processes $p_1, \dots, p_{i-1}$, and followed by processes $p_{i+1}, \dots, p_n$.

The algorithm is based on an *ordered heartbeat* monitoring model. At any time, at least one process is sending I-AM-ALIVE messages periodically to a subset of the processes in the system. Processes monitor each other by waiting for these periodical I-AM-ALIVE messages. To monitor a process p_j, process p_i uses an estimated time value that tells how much time it has to wait for the I-AM-ALIVE message from p_j. This time value is denoted by $\Delta_{i,j}$. Then, if after $\Delta_{i,j}$ time p_i did not receive the I-AM-ALIVE message from p_j, it suspects that p_j has crashed. We need to allow these time values to vary over time in the algorithm. We use $\Delta_{i,j}(t)$ to denote the value of $\Delta_{i,j}$ at time t.

Figure 4.6 presents the algorithm that is the core of the $\Diamond\mathcal{S}$ failure detector implementation. This algorithm, when run by each process of the system, guarantees that eventually all the correct processes agree on a common correct process, denoted p_{leader}. This property trivially allows us to provide the eventual weak accuracy property required by $\Diamond\mathcal{S}$: eventually, p_{leader} is not suspected by any correct process. The strong completeness property of $\Diamond\mathcal{S}$ is reached by simply making every process p_i suspect all processes in the system except p_{leader}.

Each process p_i runs an instance of the algorithm of Figure 4.6, in which there is a local variable $trusted_i$. As we will show, eventually the value of $trusted_i$ for each correct process p_i will be the same, and $p_{trusted_i}$ will be the correct process p_{leader}. In fact, $trusted_i = leader$ will be the index of the first process $\in$ *correct* in the system (following the order $p_1, ..., p_n$). From the value of $trusted_i$, it can be derived the set $suspected_i$, which will satisfy the accuracy and completeness properties of $\Diamond\mathcal{S}$. Here we consider two possibilities, to make $suspected_i = \Pi - \{p_{trusted_i}\}$ or $suspected_i = \Pi - \{p_{trusted_i}, p_i\}$, depending on whether we want to preserve the intuitive notion of a process never suspecting itself.

The algorithm of Figure 4.6 executes as follows. Initially, each process p_i starts with $trusted_i = 1$, which means that p_1 will be their first candidate to be the process p_{leader}[5]. Process p_1 starts sending I-AM-ALIVE messages periodically to every other process $p_2, \ldots, p_n$. In general, a process p_i will be sending I-AM-ALIVE messages periodically to its successors $p_{i+1}, \ldots, p_n$ iff $i = trusted_i$ (Task 1). A process p_i such that $trusted_i \neq i$, just waits for periodical I-AM-ALIVE messages from the process $p_{trusted_i}$. If it does not receive an I-AM-ALIVE message on time (within some timeout period $\Delta_{i,trusted_i}$), then it suspects that $p_{trusted_i}$

[5]The algorithm is also correct if initially each process p_i starts trusting a process chosen at random among the processes $p_1, \ldots, p_i$.

has crashed and chooses the next candidate to be the process p_{leader} by increasing $trusted_i$ by one (Task 2).

If, later on, a process p_i receives an I-AM-ALIVE message from a process p_j, such that $j < trusted_i$, then p_i will stop assuming that p_j has crashed, and will trust p_j again (by making $trusted_i = j$). In order to prevent this from happening an infinite number of times, p_i also increases the value of the timeout period $\Delta_{i,j}$ (Task 3). Moreover, if p_i was sending I-AM-ALIVE messages periodically, it will automatically stop sending them, since now $i \neq trusted_i$.

4.5.2 Detailed Algorithm and Proof of Correctness

```
Every process p_i, i = 1, ..., n executes:

trusted_i ← 1
∀j ∈ {1, ..., i − 1} : Δ_{i,j} ← default timeout
received_i ← false

cobegin

|| Task 1:
  loop
    if i = trusted_i then
      send I-AM-ALIVE to p_{i+1}, ..., p_n
    end if
    delay Δ_{HEARTBEAT}
  end loop

|| Task 2:
  loop
    if trusted_i < i then
      delay Δ_{i,trusted_i}
      if received_i then
        received_i ← false
      else
        trusted_i ← trusted_i + 1
      end if
    end if
  end loop

|| Task 3:
  loop
    receive I-AM-ALIVE from p_j
    if j = trusted_i then
      received_i ← true
    else if j < trusted_i then
      Δ_{i,j} ← Δ_{i,j} + 1
      trusted_i ← j
      received_i ← true
    else
      discard message
    end if
  end loop

coend
```

Figure 4.6: Algorithm used to implement a failure detector of class $\Diamond\mathcal{S}$.

We now show that the algorithm of Figure 4.6, combined with either of the proposed definitions of $suspected_i$ ($\Pi - \{p_{trusted_i}\}$ or $\Pi - \{p_{trusted_i}, p_i\}$), implements a failure detector of class $\diamond\mathcal{S}$. Among the processes $p_1, \ldots, p_n$, let p_{leader} be the correct process with the smallest index. The key of the proof is to show that, eventually and permanently, $trusted_i = leader$ for every correct process p_i. Thus, with either definition of $suspected_i$, eventually some correct process (namely p_{leader}) is never suspected by any correct process, which provides the eventual weak accuracy property required by $\diamond\mathcal{S}$, and eventually all crashed processes are permanently suspected by all correct processes, which provides the strong completeness property required by $\diamond\mathcal{S}$.

All time instants considered in the rest of this section are assumed to be after GST (Global Stabilization Time). We also assume that, at these instants, all messages sent before GST have already been delivered and processed. These assumptions allow us to consider in the rest of the section that the unknown bounds on processor relative speeds and on message transmission time hold. We denote by $trusted_i(t)$ the value of $trusted_i$ at time t.

Lemma 7 *$\exists t_0 : \forall t > t_0, \forall p_i \in correct, trusted_i(t) \geq leader$.*

Proof: Let p_i be any correct process. By definition of p_{leader}, eventually all its predecessors, $p_1, \ldots, p_{leader-1}$, will crash. Consider a time t' at which all the predecessors of p_{leader} have crashed and all their messages have already been delivered and processed (in Task 3). Then, if $trusted_i(t') < leader$, at most at time $t'' = t' + \Delta_{i,trusted_i}(t')$ the variable $received_i$ will take the value $false$ (see Task 2). By Task 3, $received_i$ will not become $true$ until $trusted_i \geq leader$. Therefore, at most at time $t_i = t'' + \sum_{j=1}^{leader-1} \Delta_{i,j}(t')$ the variable $trusted_i$ will take a value greater or equal to $leader$ (see Task 2). Since p_i will never receive after t_i any other message from processes $p_1, \ldots, p_{leader-1}$, the variable $trusted_i$ will never take a value below $leader$ (see Task 3).

Let $t_0 = \max\{t_i : p_i \in correct\}$. From the above reasoning, $\forall t > t_0$, $\forall p_i \in correct$, $trusted_i(t) \geq leader$. ■

Lemma 8 *$\forall t > t_0$, where t_0 is the same as in Lemma 7, $trusted_{leader}(t) = leader$.*

Proof: Note from Task 2 that $\forall t : trusted_{leader}(t) \leq leader$. Also, from Lemma 7, $\forall t > t_0$,

$trusted_{leader} \geq leader$. Hence, $\forall t > t_0$, $trusted_{leader} = leader$. ∎

Lemma 9 *After t_0, the process p_{leader} will be permanently sending* I-AM-ALIVE *messages periodically to all its successors $p_{leader+1}, \ldots, p_n$.*

Proof: It follows from Lemma 8 and Task 1. ∎

Lemma 10 *Let $p_i \in correct : i \neq leader$. If at time $t' > t_0$, $trusted_i(t') > leader$, then $\exists t''$: $t' < t'' \leq t' + \Delta_{HEARTBEAT} + \Delta_{msg}$ and $trusted_i(t'') = leader$.*

Proof: Note that, by definition of p_{leader}, p_i has to be a successor of p_{leader}. From Lemma 9, after time t_0 the process p_{leader} is permanently sending I-AM-ALIVE messages, with a period of $\Delta_{HEARTBEAT}$, to all its successors, including p_i. After t', the first I-AM-ALIVE message will be sent by p_{leader} at time $t' + \Delta_{HEARTBEAT}$ at the latest. Note that, since we have a partially synchronous system, this message takes a maximum time of Δ_{msg} to be delivered and processed by p_i. Hence, at some time $t'' \leq t' + \Delta_{HEARTBEAT} + \Delta_{msg}$, p_i will deliver and process an I-AM-ALIVE message from p_{leader}. From Lemma 7, $trusted_i \geq leader$ at t'', and then from Task 3, $trusted_i$ will take the value $leader$ at that time. ∎

Lemma 11 *Let $p_i \in correct : i \neq leader$. After t_0, $trusted_i$ will change from $leader$ to a value different from $leader$ a finite number of times.*

Proof: Let us assume, by the way of contradiction, that $trusted_i$ changes from $leader$ to a value different from $leader$ an infinite number of times. From Lemma 10, the value of $trusted_i$ will be $leader$ at some time after t_0. From Task 2, $trusted_i$ changes from $leader$ to $leader + 1$ if two I-AM-ALIVE messages are received by p_i more than $\Delta_{i,leader}$ time apart. Note from Task 1 and from the fact that we have a partially synchronous system that two consecutive I-AM-ALIVE messages sent by p_{leader} are received and processed by p_i at most $\Delta_{HEARTBEAT} + \Delta_{msg}$ time apart. Also, from Lemma 10, the value of $trusted_i$ will become $leader$ again eventually. Every time this happens, from Task 3, the value of $\Delta_{i,leader}$ is incremented by one. Hence, since this will happen an infinite number of times, eventually $\Delta_{i,leader}$

will be larger than $\Delta_{HEARTBEAT} + \Delta_{msg}$. However, after that happens $trusted_i$ will never change its value from $leader$, which is a contradiction. ■

Theorem 8 $\exists t_1 : \forall t > t_1, \forall p_i \in correct, trusted_i(t) = leader.$

Proof: Follows from Lemma 8 for the case $i = leader$, and from Lemma 10 and Lemma 11 for the case $i \neq leader$. ■

Corollary 14 *Let $suspected_i$ be defined as either $\Pi - \{p_{trusted_i}\}$ or $\Pi - \{p_{trusted_i}, p_i\}$, $\forall p_i \in \Pi$. The algorithm of Figure 4.6, combined with either of these definitions of $suspected_i$, implements a failure detector of class $\diamond\mathcal{S}$.*

4.5.3 Performance Analysis

In this section, we first evaluate the performance of the presented ordered-heartbeat algorithm in terms of the number and size of the messages periodically sent. This evaluation shows that this algorithm compares favorably with the algorithm proposed by Chandra and Toueg of Section 4.3 and the ring-based algorithms of Section 4.4.

In general, since algorithms implementing failure detectors need not necessarily be periodic, it is not always possible to evaluate their efficiency in terms of the amount of information periodically sent. For this reason, in order to evaluate the efficiency of this kind of algorithms independently of the communication pattern they use, we propose a new performance measure, which we call *eventual monitoring degree*. Informally, this measure consists on the number of pairs of correct processes that will infinitely often communicate. We show that the runs of the ordered-heartbeat algorithm of Figure 4.6 are optimal, and at least as good as the runs of any algorithm implementing a failure detector of class $\diamond\mathcal{W}$, with respect to this measure.

It is well known that, in practice, every system goes through *stable* and *unstable* periods. Informally, a stable period is a period during which no process crashes and there are no timing failures, i.e., all messages arrive on time. On the other hand, an unstable period is a period during which processes may crash or there are timing failures, i.e., messages can be lost or arrive late. If we assume that the system is most frequently in stable periods, then the eventual monitoring degree is still a valid performance measure within each of these stable periods.

Number and size of messages and amount of information

Under the same assumptions made as in Section 4.4.6, the number of messages periodically sent in the algorithm of Figure 4.6 is eventually at most $n-1$, since eventually only the process p_{leader} will be sending I-AM-ALIVE messages periodically to its successors. In Chandra-Toueg's algorithm implementing $\diamond\mathcal{P}$ of Section 4.3, the number of messages periodically sent eventually becomes $n\mathcal{C}$, since eventually only the $\mathcal{C}$ correct processes will be sending messages periodically to every other process. Finally, in the ring-based algorithms implementing $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ of Section 4.4, the number of messages periodically sent is eventually $2\mathcal{C}$, since eventually each correct process communicates with its predecessor and its successor in the ring formed by the correct processes. Note that the algorithm of Figure 4.6 requires less messages than Chandra-Toueg's. Compared with the ring-based algorithms, if the probability of process failure is low, the value of $\mathcal{C}$ will be close to n and the ordered-heartbeat algorithm will also require less messages.

However, note that only $\mathcal{C}-1$ messages out of the $n-1$ periodically sent by p_{leader} will be actually delivered and processed[6]. Similarly, the actual number of messages delivered and processed in Chandra-Toueg's algorithm is $\mathcal{C}^2$.

Concerning the size of the messages, the ordered-heartbeat algorithm requires messages of $\Theta(\log n)$ bits, since each message needs to carry the identity of its sender. This is also the case for Chandra-Toueg's algorithm. However, the ring-based algorithms require messages of $\Theta(n)$ bits, since messages carry a list of suspected processes.

If we look now at the total amount of information periodically sent, the ordered-heartbeat algorithm eventually requires $\Theta(n \log n)$ bits to be sent. The total amount of information periodically sent by Chandra-Toueg's algorithm is eventually $\Theta(n\mathcal{C} \log n)$ bits. Finally, the ring-based algorithms eventually require $\Theta(n\mathcal{C})$ bits to be sent.

[6]The amount of processing is better captured with the *eventual monitoring degree* measure, considered later in this section.

Algorithm	Number of messages	Size of messages	Amount of information	Eventual monitoring degree
Chandra-Toueg ($\Diamond\mathcal{P}$)	$n\mathcal{C}$	$\Theta(\log n)$	$\Theta(n\mathcal{C}\log n)$	C^2
Ring-based ($\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$)	$2\mathcal{C}$	$\Theta(n)$	$\Theta(n\mathcal{C})$	$2\mathcal{C}$
Ordered-heartbeat ($\Diamond\mathcal{S}$)	$n-1$	$\Theta(\log n)$	$\Theta(n\log n)$	$\mathcal{C}-1$

Figure 4.7: Evaluation of the different algorithms implementing failure detectors.

Eventual monitoring degree

It must be clear that, in general, different algorithms can use a different period to exchange information between processes. Furthermore, this exchange of information in a given algorithm need not be periodic. In this set up, it is difficult to use the number of messages periodically exchanged as the efficiency measure of this kind of algorithms. For this reason, on top of the number and size of messages periodically sent, we will also use the *eventual monitoring degree* measure to evaluate the efficiency of the algorithms, which we define as follows.

Definition 1 *The* eventual monitoring degree *of a run R of an algorithm A is the number of pairs (p_i, p_j) of correct processes, such that p_i sends infinitely many messages to p_j in run R.*

We consider this measure more adequate to evaluate the efficiency of algorithms implementing failure detectors, since it is independent of the communication pattern they use, in particular of their potential periodicity. Moreover, in practice, this measure is close to the number of messages exchanged during a stable period.

We now evaluate the algorithms in terms of the *eventual monitoring degree* of their runs, as defined in Definition 1. It is easy to observe that the eventual monitoring degree of any run of the ordered-heartbeat algorithm of Figure 4.6 is $\mathcal{C}-1$, since eventually only the correct processes different from p_{leader} will receive messages, and they will receive them only from p_{leader}.

The eventual monitoring degree of any run of Chandra-Toueg's algorithm implementing $\Diamond\mathcal{P}$ is $\mathcal{C}^2$, since eventually each correct process will receive messages from every correct process. Similarly, the eventual monitoring degree of any run of the ring-based algorithms implementing $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ is $2\mathcal{C}$, since eventually each correct process will only receive messages from its predecessor and its successor in the ring formed by the correct processes.

Figure 4.7 summarizes the evaluation of the different algorithms implementing failure detectors.

We now show that any run of the algorithm of Figure 4.6 is optimal with respect to the eventual monitoring degree measure, i.e., any run of the algorithm has the minimum eventual monitoring degree needed to implement the weakest failure detector for solving Consensus. For generality, we will consider the class $\diamond\mathcal{W}$ of failure detectors, showing that the eventual monitoring degree required by a run of any algorithm implementing a failure detector of class $\diamond\mathcal{W}$ is at least $\mathcal{C} - k$, where k is the assumed minimum number of correct processes in the system. Observe that, by definition, $\mathcal{C} \geq k$. Note also that up to now we have assumed a system with $k = 1$. However, in other system models it is assumed a larger value of k (for instance, a majority of correct processes, i.e., $k > n/2$). We use the parameter k here for the generality of the lemma.

Lemma 12 *Any run R of an algorithm A implementing a failure detector of class $\diamond\mathcal{W}$ has an eventual monitoring degree of at least $\mathcal{C}_R - k$, where $\mathcal{C}_R$ is the number of correct processes in R, and k is the assumed minimum number of correct processes in the system.*

Proof: The claim trivially holds when $\mathcal{C}_R = k$. Let us assume now that $\mathcal{C}_R > k$. We will prove the claim by contradiction, showing that if R does not satisfy the claim (i.e., its eventual monitoring degree is less than $\mathcal{C}_R - k$), then we can find another run R' of algorithm A that violates some property of $\diamond\mathcal{W}$.

By the way of contradiction, let us assume that the eventual monitoring degree of the run R is at most $\mathcal{C}_R - k - 1$. Hence, there is some time t and there are at least $k + 1$ correct processes such that these processes do not receive messages from any other process after time t. From the fact that A implements a failure detector of class $\diamond\mathcal{W}$, the eventual weak accuracy property is satisfied in R. Therefore, there is a time t' and a correct process p_j such that p_j is never suspected by any correct process after t'.

Let us consider now another run R' of A, which behaves exactly the same as R up to a time instant $t'' > max(t, t')$. Let $p_1, \ldots, p_k$ be k correct processes distinct from p_j which do not receive messages from any other process after time t. Let us assume that in R' all processes except $p_1, \ldots, p_k$ crash at time t'', and $p_1, \ldots, p_k$ are correct. Clearly, process p_j crashes in R' and, by the weak completeness property of A, p_j should be suspected by at least one of the correct processes $p_1, \ldots, p_k$. Note that in R', after time t'' no message is received by $p_1, \ldots, p_k$ from any other process, as it happened in R. Hence, from the point of view of processes $p_1, \ldots, p_k$, both runs R and R' are indistinguishable. Therefore, processes $p_1, \ldots, p_k$ take the same decisions in both runs. In particular, after time t' none of $p_1, \ldots, p_k$ will suspect process p_j in R', as they do in R. This violates the weak completeness property in R' and, thus, A does not implement a failure detector of class $\diamond\mathcal{W}$, which is a contradiction. ■

Theorem 9 *For $k = 1$, where k is the assumed minimum number of correct processes in the system, any run of the algorithm of Figure 4.6 has an optimal eventual monitoring degree of $\mathcal{C} - 1$.*

Proof: Follows from Lemma 12 and the observation made at the beginning of this section (i.e., the eventual monitoring degree of any run of the algorithm of Figure 4.6 is $\mathcal{C} - 1$). ■

4.6 A Note on Communication Failures

All the algorithms proposed in this chapter assumed that processes exchange messages through quasi-reliable communication channels. If we now consider that communication channels are unreliable up to GST, i.e., messages sent before GST can be lost, it is easy to see that the algorithms based on the heartbeat monitoring model, e.g., Chandra-Toueg's algorithm of Section 4.3 and the ordered-heartbeat algorithm of Section 4.5, are still correct. Thus, the heartbeat monitoring model can be viewed as a reliability mechanism, equivalent to message retransmission.

On the other hand, the algorithms based on the polling monitoring model, e.g., the ring-based algorithms of Section 4.4 are no longer valid if communication channels are unreliable up to GST. For instance, all ARE-YOU-ALIVE? messages sent before GST could be lost and the system will incorrectly stabilize in a scenario in which each process suspects every other process.

4.7 Empirical Results by Simulation

In this section, we present the results of evaluating the different algorithms implementing failure detectors using discrete event simulation. We have evaluated the six algorithms presented in this chapter, namely Chandra-Toueg's algorithm implementing $\diamond\mathcal{P}$, the ring-based algorithms implementing $\diamond\mathcal{W}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, and $\diamond\mathcal{P}$, and the ordered-heartbeat algorithm implementing $\diamond\mathcal{S}$.

The parameters characterizing a simulation are the following:

- **The duration of the simulation.** 100.000.000 time units in all cases.
- **The number of processes.** We have made simulations with 5, 10, 15, 20 and 25 processes.
- **The failure pattern, i.e., which processes crash and when.** We consider a different failure pattern for each different number of processes. However, for a given number of processes, the failure pattern is the same for all the algorithms.
- **The transmission time of a message.** Random value between 50 and 75 time units[7].
- **The delay between the sending of two heartbeats.** This parameter is necessary only for the algorithms based on the heartbeat monitoring model. We have set this parameter to a fixed value of 100 time units. Obviously, it is not convenient to set this parameter to a value below the maximum transmission time of a message.
- **The initial timeout values.** We take as initial timeout values the minimum transmission time for algorithms based on heartbeats, i.e., 50 time units, and two times this value for algorithms based on pollings, i.e., 100 time units.

As expected from the previous values, the failure detectors stabilize when the timeouts reach a value of $100 + 25 = 125$ time units in the case of the algorithms based on heartbeats, and a value of $2 * 75 = 150$ time units in the case of the algorithms based on polling. It is important to note that the objective of the simulations was uniquely *to compare the behavior of the different algorithms under the same conditions*, i.e., the same parameter values. The

[7] The values have been calculated assuming a 10 Mb Ethernet.

measures obtained by simulation have been the total number of messages exchanged, the latency of the failure information propagation, and the number of mistakes (incorrect suspicions) made by each failure detector. In general, the results that have been obtained confirm the analytical performance evaluation made in previous sections.

In all the figures that follow, CT denotes Chandra-Toteg's algorithm implementing $\diamond\mathcal{P}$, LAF denotes one of the ring-based algorithms (with indication of the failure detector class that is implemented as a subscript), and LFA denotes the ordered-heartbeat algorithm implementing $\diamond\mathcal{S}$.

4.7.1 Number of Messages and Eventual Monitoring Degree

Figures 4.8 and 4.9 show the total number of messages exchanged by the algorithms and their eventual monitoring degree respectively. The total number of messages has been obtained by simulation. On the other hand, the eventual monitoring degree has been obtained analytically[8], considering only executions without crashes. Note the similarities between both figures. As expected, Chandra-Toueg's algorithm has a quadratic complexity, while the ring-based algorithms and the ordered-heartbeat algorithms have a linear complexity. Finally, the ordered-heartbeat algorithm has the best performance in both measures.

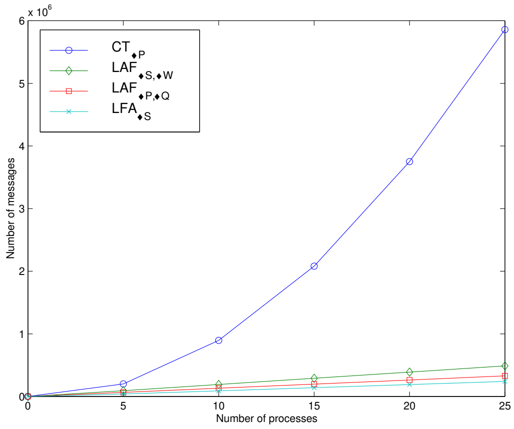

Figure 4.8: Number of messages.

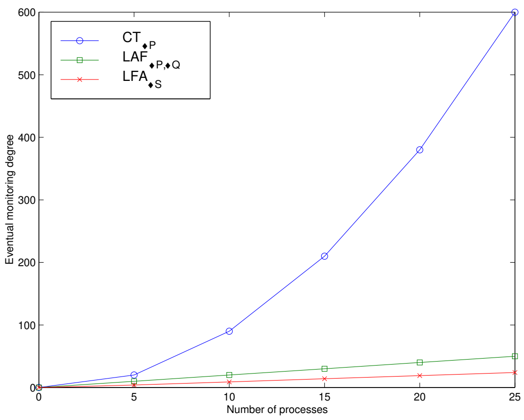

Figure 4.9: Eventual monitoring degree.

[8] $n * (n-1)$ in the case of CT, $2 * n$ in the case of LAF, and $n-1$ in the case of LFA.

4.7.2 Latency of Failure Information Propagation

Figure 4.10 shows the latency of the failure information propagation for the failure detectors satisfying strong completeness (we do not consider the ordered-heartbeat algorithm). We have measured the time elapsed between the instant in which one process detects the crash of a given process p and the instant in which all the processes (except those that have crashed) detect the crash of p. As expected, in the case of Chandra-Toueg's algorithm the latency is practically constant (with a value close to 100 time units), while in the case of the ring-based algorithms the latency increases linearly with the number of processes. This is a consequence of the message exchange pattern of the algorithms (all-to-all in CT vs. one-to-one over the ring in LAF). Note also that the latency is bigger in the case of the ring-based implementation of $\diamond\mathcal{P}$ than in the case of the ring-based implementation of $\diamond\mathcal{S}$. The reason is that in the former all timeouts between pairs of processes over the ring are increased in order to achieve eventual strong accuracy (see figure 4.4), while in the later only few ones are increased to achieve eventual weak accuracy (see Figure 4.3). As a consequence, pollings are more frequent in the case of $\diamond\mathcal{S}$, which makes the information about failures to be propagated faster.

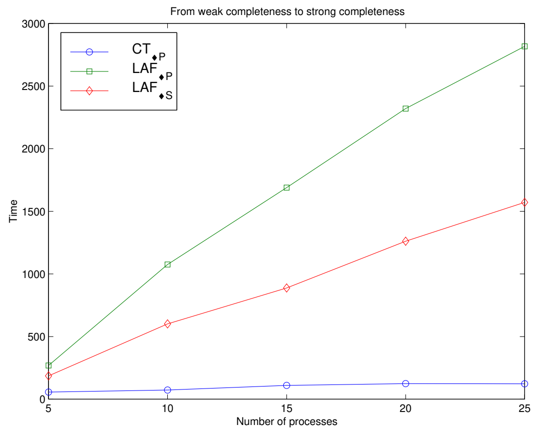

Figure 4.10: Latency of failure information propagation.

4.7.3 Number of Incorrect Suspicions

Figures 4.11 to 4.15 show the evolution of the number of incorrect suspicions made by the failure detectors satisfying eventual strong accuracy. Actually, we focus only on the two implementations of $\diamond\mathcal{P}$, since the results obtained for the ring-based implementations of $\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$ are almost identical. As expected, in the case of Chandra-Toueg's algorithm the number of incorrect suspicions is bigger at the beginning of the execution, but decreases faster than in the case of the ring-based implementation of $\diamond\mathcal{P}$. Once again, this is a consequence of the monitoring pattern of the algorithms (i.e., all-to-all in CT vs. one-to-successors in LAF).

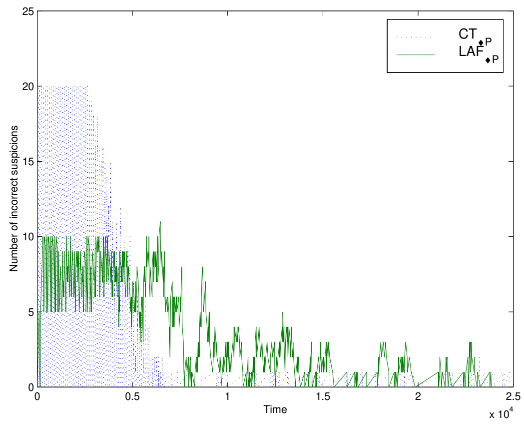

Figure 4.11: Incorrect suspicions (n = 5).

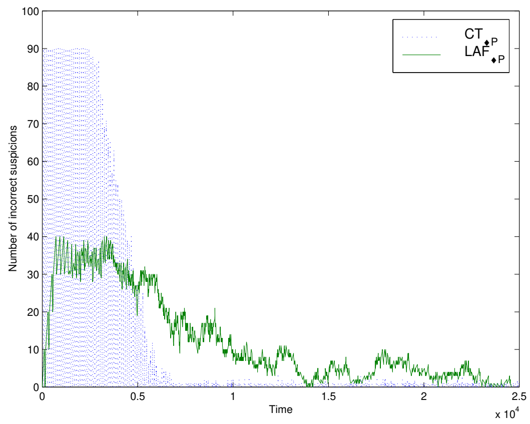

Figure 4.12: Incorrect suspicions (n = 10).

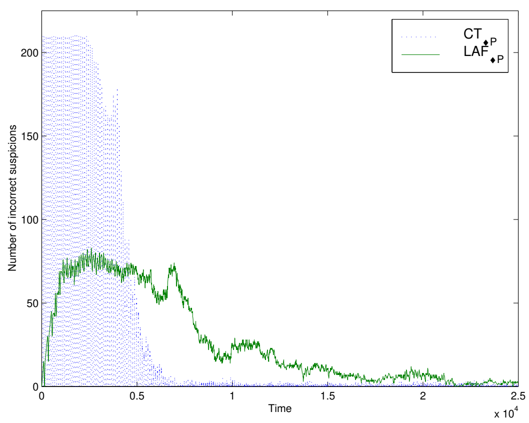

Figure 4.13: Incorrect suspicions (n = 15).

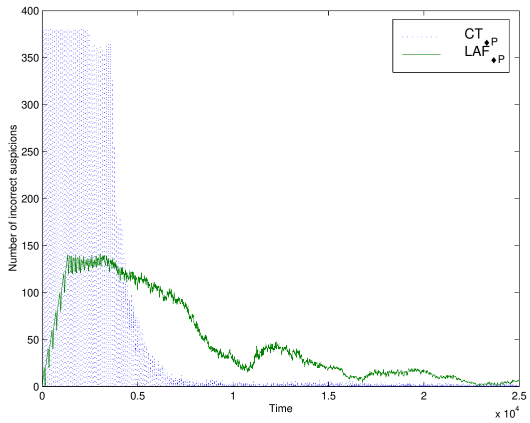

Figure 4.14: Incorrect suspicions (n = 20).

4.8 Related Work

Application Dependent Failure Detectors

In [Ser98, SDS99], Sergent et al. study the impact of different implementations of failure detectors on the termination time of the Chandra-Toueg's consensus algorithm based on $\diamond\mathcal{S}$. They first study the general heartbeat and interrogation implementations of failure detectors. As an alternative, they also propose two *consensus-dependent* implementations of failure de-

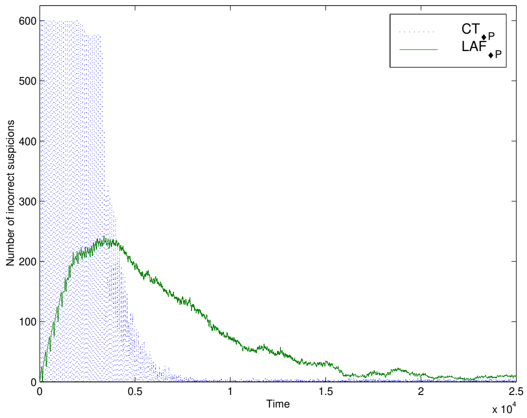

Figure 4.15: Incorrect suspicions (n = 25).

tectors. The idea is to optimize the failure detector by adopting an implementation which is customized for the consensus algorithm. By using algorithm-dependent implementations, one can reduce the number of messages generated by the failure detector. For instance, one of their implementations (called "no-message") does not require the sending of any message (it is based on starting a timer, and waiting for a message of the consensus algorithm). The other implementation ("ad-hoc heartbeat") consists in the coordinator of each round sending periodical heartbeats in a given moment of the round, in order to prevent erroneous suspicions of the coordinator that appeared frequently with the "no-message" implementation.

Clearly, these two implementations rely on the consensus algorithm which is executed on top of the failure detector, and thus cannot be taken as general implementations, nor be shared by other algorithms. A general implementation of the failure detector has the advantage that it can be used – even concurrently – by different algorithms (i.e., its cost is amortized).

Gossip-style Failure Detector

In [vRMH98], van Renesse et al. present a scalable gossip-style randomized failure detector protocol. They measure the accuracy of this protocol in terms of the *probability of mistakes*, i.e, the probability that when a timer is set, it will prematurely timeout on a process that is actually up. In gossip protocols, unlike in traditional protocols, a process sends messages to *randomly chosen* processes. In this protocol, each process maintains a list of values that indicates for each process a strictly increasing heartbeat counter. Processes occasionally send their list to randomly chosen processes, or broadcast it to all processes. Upon reception of a list, a process merges the old and the new lists by keeping the maximum heartbeat counter of each process.

On the Quality of Service of Failure Detectors

In [Che00, CTA00], Chen et al. study the *quality of service (QoS)* of failure detectors, i.e., (a) how fast the failure detector detects actual failures, and (b) how well it avoids false detections. They first propose a set of basic QoS metrics to specify failure detectors for systems where message delays and message losses follow some probability distributions. Informally, these metrics are the following:

- *detection time* of a crashed process, i.e., the time elapsed from the moment when a process p crashes to the time when the failure detector starts suspecting p permanently.
- *mistake recurrence time*, i.e., the time between two consecutive mistakes[9].
- *mistake duration*, i.e., the time it takes the failure detection to correct a mistake.

Then, they give a new failure detector algorithm for a basic system consisting on two processes in which one of them monitors the other, and show that it is optimal with respect to some of the proposed metrics. This algorithm assumes that processes have synchronized clocks[10]. It is worth to note that the ordered-heartbeat algorithm proposed in Section 4.5 can benefit from the techniques described in this works.

$\infty\infty\infty$

[9] The failure detector makes a mistake each time its output changes from "not suspect" to "suspect" while p is actually up.

[10] When synchronized clocks are not available, they propose a modification of the algorithm that performs equally well in practice.

Chapter 5

Eventually Consistent Failure Detectors

5.1 Introduction

In this chapter, we propose a new class of unreliable failure detectors, which we call *Eventually Consistent* and denote by $\Diamond\mathcal{C}$. This class adds to the failure detection capability of other classes an *eventual leader election* capability. Informally, this leader election capability allows all correct processes to eventually choose the same correct process as leader.

We study the relationship between $\Diamond\mathcal{C}$ and the four classes of failure detectors satisfying eventual accuracy defined by Chandra and Toueg in [CT96]. We first show that $\Diamond\mathcal{P}$ is a subclass of $\Diamond\mathcal{C}$, i.e., any algorithm implementing $\Diamond\mathcal{P}$ implements also $\Diamond\mathcal{C}$. Similarly, we show that $\Diamond\mathcal{C}$ is a subclass of $\Diamond\mathcal{S}$ (and hence of $\Diamond\mathcal{W}$). Moreover, we show that $\Diamond\mathcal{C}$ and $\Diamond\mathcal{S}$ are equivalent. This means that we can obtain a failure detector of class $\Diamond\mathcal{C}$ from any failure detector of class $\Diamond\mathcal{S}$ (maybe executing a transformation algorithm).

We show that $\Diamond\mathcal{C}$ can be implemented as efficiently as $\Diamond\mathcal{S}$ in partially synchronous systems. To show it, we present some algorithms implementing $\Diamond\mathcal{S}$ that *directly* implement $\Diamond\mathcal{C}$, i.e., we do not need to execute any transformation algorithm. In particular, the two algorithms implementing $\Diamond\mathcal{S}$ presented in Chapter 4, i.e., the ring-based algorithm and the ordered-heartbeat algorithm, implement also a failure detector of class $\Diamond\mathcal{C}$.

To show the power of this new class of failure detectors, we also propose a Consensus algorithm based on an eventually consistent failure detector. This algorithm successfully exploits the leader election capability of the failure detector and, in general, performs better in the number of rounds than all the previously proposed Consensus algorithms for failure detectors with eventual accuracy. This is due to the fact that, to our knowledge, it is the first Consensus algorithm for failure detectors with eventual accuracy that does not rely on the rotating coordinator paradigm.

5.2 The Eventually Consistent Failure Detector Class, $\diamond\mathcal{C}$

In this section, we introduce the *Eventually Consistent* class of failure detectors, $\diamond\mathcal{C}$. The main characteristic of these failure detectors is the accuracy property they satisfy, which we call *Eventual Consistent Accuracy*. Informally, the eventual consistent accuracy guarantees that there is a correct process p that is eventually and permanently not suspected by any correct process, and that there is a function that each correct process can apply to the output[1] of its local failure detection module that eventually and permanently returns p.

Let us denote by $\mathcal{D}_p$ the set of suspected processes returned by a failure detector $\mathcal{D}$ to a given process p. We also denote by $\mathcal{T}_p$ the set of *trusted* (i.e., non-suspected) processes of the failure detection module attached to process p, i.e., $\mathcal{T}_p = \Pi - \mathcal{D}_p$. The eventual consistent accuracy property is defined as follows. Let $\mathcal{P}(\Pi)$ be the power set of the set Π.

Definition 2 *A failure detector $\mathcal{D}$ satisfies* Eventual Consistent Accuracy *if there is a deterministic function $leader : \mathcal{P}(\Pi) \to \Pi$, a time t and a correct process p such that, after t, for every correct process q, $p \notin \mathcal{D}_q$ and $leader(\mathcal{T}_q) = p$ forever.*

Definition 3 *A failure detector is said to be* Eventually Consistent *if it satisfies strong completeness[2] and eventual consistent accuracy. The set of all such failure detectors, called the* class of Eventually Consistent failure detectors, *is denoted by $\diamond\mathcal{C}$.*

A failure detector of class $\diamond\mathcal{C}$ enhances the classical failure detection properties of previously defined classes [CT96] with an eventual leader election mechanism. These failure detectors guarantee that after some point in time all correct processes can behave as a consistent leader election algorithm. (However, they do not give knowledge of when the leader has

[1] Actually, the function is applied to the set of non-suspected processes. If the failure detector outputs a set of suspected processes, this is just the complement of that set.

[2] *Eventually every process that crashes is permanently suspected by every correct process.*

been elected.) This property can be used by algorithms in which the safety properties are not affected by the simultaneous existence of several leaders, and that guarantee termination if a unique leader exists. Furthermore, these failure detectors can be very useful to algorithms that have early termination when there is a unique leader. As it is well known, usually it is not necessary for the failure detector to reach permanent stability to be useful. Instead, many algorithms can successfully complete if the failure detector is stable (i.e., provides a unique leader) for long enough periods of time.

Considering the Consensus problem, this property of the eventually consistent failure detectors allows every correct process to eventually agree on a coordinator that can be used to reach consensus. Hence, with these failure detectors we do not need to rely on the rotating coordinator paradigm to eventually choose an appropriate coordinator.

5.3 Relation between $\diamond\mathcal{C}$ and other Failure Detector Classes

With an eventually consistent failure detector, from the eventual consistent accuracy, eventually there is a correct process that will be never suspected by any correct process, and from the strong completeness, eventually all the crashed processes are permanently suspected by every correct process. From this, it is simple to see that every failure detector of class $\diamond\mathcal{C}$ belongs also to the class $\diamond\mathcal{S}$ (and $\diamond\mathcal{W}$).

Note also that any failure detector of class $\diamond\mathcal{P}$ belongs to the class $\diamond\mathcal{C}$ as well. With $\diamond\mathcal{P}$, eventually the set of suspected processes by every correct process becomes the same, containing only all the processes that actually crash. A possible *leader* function could always choose the first (with respect to the order $p_1, \ldots, p_n$ assumed in the system model) non-suspected process. Figure 5.1 summarizes the relation between failure detector classes $\diamond\mathcal{W}$, $\diamond\mathcal{S}$, $\diamond\mathcal{C}$, and $\diamond\mathcal{P}$.

Equivalence between $\diamond\mathcal{C}$ and Ω

To prove that $\diamond\mathcal{W}$ is the weakest failure detector for solving Consensus, Chandra et al. defined in [CHT96] a new failure detector, denoted Ω.[3] The output of the failure detector

[3] Actually, to prove their result Chandra et al. show first that Ω is at least as strong as $\diamond\mathcal{W}$, and then that any failure detector $\mathcal{D}$ that can be used to solve Consensus is at least as strong as Ω (and hence at least as strong as $\diamond\mathcal{W}$).

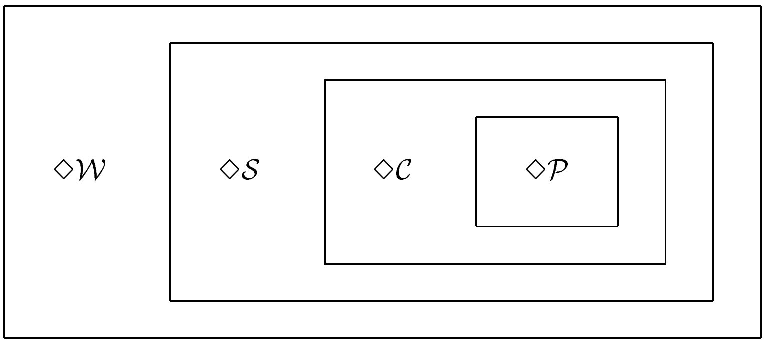

Figure 5.1: Relation between failure detector classes $\diamond\mathcal{P}$, $\diamond\mathcal{C}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$.

module of Ω at a process p is a *single* process q, that p currently considers to be *correct*; we say that p *trusts* q. The failure detector Ω satisfies the following property:

- There is a time after which all the correct processes always trust the same correct process.

As with $\diamond\mathcal{W}$, the output of the failure detector module of Ω at a process p may change with time, i.e., p may trust different processes at different times. Furthermore, at any given time t, two processes p and q may trust different processes.

It is straightforward that failure detectors $\diamond\mathcal{C}$ and Ω are equivalent, i.e., one can be transformed into the other and vice versa. In some sense, the class $\diamond\mathcal{C}$ can be viewed as a redefinition of Ω in terms of lists of suspected processes instead of a single trusted process. Nevertheless, the class $\diamond\mathcal{C}$ does not force each failure detector module to systematically suspect all the processes in the system except one, while this is the only way to achieve strong completeness having Ω.[4] The following lemma shows the equivalence between Ω and $\diamond\mathcal{C}$.

Lemma 13 *Failure detector classes Ω and $\diamond\mathcal{C}$ are equivalent.*

Proof: To transform Ω into $\diamond\mathcal{C}$: every process p queries its local failure detection module Ω_p and suspects every process except its trusted process. To transform $\diamond\mathcal{C}$ into Ω: every process p queries its local failure detection module $\diamond\mathcal{C}_p$ and applies the *leader* function to the

[4] Optionally two processes, if apart the trusted process output by Ω, each failure detector module does not suspect the process it is attached to.

complementary set $\Pi - \diamond\mathcal{C}_p$. The resulting process is output by p as its trusted process. ■

5.3.1 Equivalence between $\diamond\mathcal{C}$ and $\diamond\mathcal{S}$

We now show that any failure detector of class $\diamond\mathcal{S}$ can be transformed into a failure detector of class $\diamond\mathcal{C}$. Since, by definition, $\diamond\mathcal{C}$ is a subclass of $\diamond\mathcal{S}$, i.e., every failure detector in $\diamond\mathcal{C}$ is in $\diamond\mathcal{S}$, this shows that failure detector classes $\diamond\mathcal{C}$ and $\diamond\mathcal{S}$ are actually equivalent.

Lemma 14 *Failure detector classes $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent.*

Proof: This result is shown in [CT96]. To transform $\diamond\mathcal{S}$ into $\diamond\mathcal{W}$: trivial, since $\diamond\mathcal{S}$ is a subclass of $\diamond\mathcal{W}$ (both classes satisfy eventual weak accuracy, and strong completeness ($\diamond\mathcal{S}$) involves weak completeness ($\diamond\mathcal{W}$)). To transform $\diamond\mathcal{W}$ into $\diamond\mathcal{S}$: transformation protocol given in [CT96] (see Appendix A). ■

Lemma 15 *Failure detector classes $\diamond\mathcal{W}$ and Ω are equivalent.*

Proof: This result is shown in [CHT96]. To transform $\diamond\mathcal{W}$ into Ω: generic transformation protocol in the proof in [CHT96], two transformation protocols in [Chu98]. To transform Ω into $\diamond\mathcal{W}$: every process p queries its local failure detection module Ω_p and suspects every process except its trusted process[5]. ■

The *Slander* algorithm presented in Figure 5.2, proposed by Chu [Chu98], transforms any failure detector of class $\diamond\mathcal{W}$ into the failure detector Ω. The basic idea is that each process gossips about what it heard from its local failure detector module and keeps track of the number of "slanders" it heard regarding a process. The process with the least count is deemed the most trustworthy. Note that MAX in the algorithm computes the component-wise maximum of two vectors and MIN gives the *first* index that realizes the minimum.

Theorem 10 *Failure detector classes $\diamond\mathcal{C}$ and $\diamond\mathcal{S}$ are equivalent.*

[5] Identical to transforming Ω into $\diamond\mathcal{C}$.

Every process p executes the following:

$trust_p \leftarrow p$
$count_p \leftarrow \langle 0, \ldots, 0 \rangle$

cobegin

|| *Task 1:* **repeat forever**
  $\{p$ *queries its local failure detector module* $\mathcal{D}_p\}$
  $suspects_p \leftarrow \mathcal{D}_p$
  foreach $q \in suspects_p$ **do**
    $count_p[q] \leftarrow count_p[q] + 1$
  send $count_p$ to all

|| *Task 2:* **when** receive $count_q$ for some q
  $count_p \leftarrow \text{MAX}(count_p, count_q)$
  $trust_p \leftarrow \text{MIN}(count_p)$

coend

Figure 5.2: Transforming $\diamond\mathcal{W}$ into Ω. The Slander algorithm.

Proof: To transform $\diamond\mathcal{C}$ into $\diamond\mathcal{S}$: trivial, since $\diamond\mathcal{C}$ is a subclass of $\diamond\mathcal{S}$ (both classes satisfy strong completeness, and eventual consistent accuracy ($\diamond\mathcal{C}$) involves eventual weak accuracy ($\diamond\mathcal{S}$)). To transform $\diamond\mathcal{S}$ into $\diamond\mathcal{C}$: follows from Lemmas 14, 15, and 13. ■

Thus, having any failure detector of class $\diamond\mathcal{S}$, it is always possible to build a failure detector of class $\diamond\mathcal{C}$. However, instead of starting from any failure detector of class $\diamond\mathcal{S}$ and running a – maybe expensive[6] – transformation protocol on top of it in order to get a failure detector of class $\diamond\mathcal{C}$, we show in the next section that there are extremely efficient implementations of $\diamond\mathcal{C}$.

5.4 Implementations of $\diamond\mathcal{C}$

There are several algorithms implementing failure detectors of the classes defined in [CT96] that also implement eventually consistent failure detectors. For instance, since $\diamond\mathcal{P}$ is a

[6]The transformation protocols of Chandra et al. [CHT96] and Chu [Chu98] are quadratic, i.e., they require that every process sends messages periodically to all processes in the system.

subclass of $\diamond\mathcal{C}$ Chandra-Toueg's algorithm implementing $\diamond\mathcal{P}$ of Section 4.3, as well as the ring-based algorithm implementing $\diamond\mathcal{P}$ of Section 4.4 implement also a failure detector of class $\diamond\mathcal{C}$.

Concerning the ring-based algorithm implementing $\diamond\mathcal{S}$, the set of non-suspected processes can be different in different processes, but the implementation guarantees that eventually the first (starting from the *initial candidate to leader* and following the order defined by the ring) non-suspected process is the same for every correct process, and that it is also a correct process. From this, it is easy to derive a deterministic function *leader* that returns the first non-suspected process. Hence, the ring-based algorithm implementing $\diamond\mathcal{S}$ of Section 4.4 implements also a failure detector of class $\diamond\mathcal{C}$.

Let us consider now the optimal ordered-heartbeat algorithm implementing $\diamond\mathcal{S}$ of Section 4.5. With this algorithm, the set of non-suspected processes contains only one process, that will eventually and permanently be the same correct process for all correct processes[7], which trivially gives us a possible *leader* function. Again, this failure detector implements also a failure detector of class $\diamond\mathcal{C}$.[8]

5.5 The Consensus Problem

In the Consensus problem, each process proposes initially a value, and all correct processes must reach an irrevocable decision on some common value that is equal to one of the proposed values [Fis83, CT96]. Formally, the Consensus problem is defined in terms of two primitives, *propose* and *decide*. When a process executes *propose*(v), we say that it *proposes* v; similarly, when a process executes *decide*(v), we say that it *decides* v. The *Consensus* problem is specified by the following properties:

- *Termination.* Every correct process eventually decides some value.
- *Uniform integrity.* Every process decides at most once.
- *Agreement.* No two correct processes decide differently.

[7]We consider the following definition for the set of suspected processes: $\Pi - \{p_{trusted_i}\}$. If we consider the other possible definition, i.e., $\Pi - \{p_{trusted_i}, p_i\}$, then the set of non-suspected processes will eventually contain two correct processes for all correct processes except p_{leader}, and only one correct process for p_{leader}. In any case, the process p_{leader} will be the *first* process in that set.

[8]Actually, the algorithm of Figure 4.6 implements Ω.

- *Validity.* If a process decides v, then v was previously proposed by some process.

Termination defines the *liveness* property of Consensus, while Uniform integrity, Agreement and Validity define its *safety* properties.

The Agreement property allows faulty processes to decide differently from correct processes. This fact can be sometimes undesirable as it does not prevent an incorrect process to propagate a different decision throughout the system before crashing. In the *Uniform Consensus* [NT90] problem, agreement is defined by the following property, which enforces the same decision on any process that decides:

- *Uniform Agreement.* No two processes (correct or faulty) decide differently.

It has been shown in [Gue95] that any algorithm that solves Consensus using a failure detector of class $\diamond\mathcal{S}$, also solves Uniform Consensus. In the previous section we have seen that every failure detector of class $\diamond\mathcal{C}$ belongs also to the class $\diamond\mathcal{S}$. Thus, in the following we consider the Uniform Consensus problem.

5.5.1 Solving Consensus using $\diamond\mathcal{S}$

All the Consensus algorithms based on unreliable failure detectors with eventual accuracy we know of require at least a failure detector of class $\diamond\mathcal{S}$ [CT96, Sch97, HR99, MR99a, HMMR00]. All the algorithms assume a majority of correct process (this is a necessary requirement for solving Consensus using a failure detector with eventual accuracy). All of them proceed in asynchronous rounds, in each of which a different process acts as coordinator. This approach is known as the *rotating coordinator paradigm.* By the eventual weak accuracy property of $\diamond\mathcal{S}$, it is guaranteed the existence of a correct process, namely *leader_process*, that is eventually not suspected by any correct process. As the role of coordinator rotates among all the processes in the system, eventually *leader_process* will become the coordinator of a round after no process suspects it, which guarantees the termination of Consensus.

The inconvenience of the $\diamond\mathcal{S}$-Consensus algorithms based on the rotating coordinator paradigm is that if *leader_process* is not coordinator until round i (where i could be $\Omega(n)$, with n the number of processes in the system), they may require i rounds to reach consensus. In [HMMR00], a Consensus algorithm is proposed that allows a process to be in several rounds simultaneously to reduce the time to reach consensus. However, it uses the rotating coordinator paradigm and, hence, suffers of the problem described above. It would be nice to have Consensus algorithms that quickly choose *leader_process* as the coordinator, hence reducing the number of rounds required to reach consensus.

The Consensus algorithms proposed in [DLS88] also use the rotating coordinator paradigm and can present the above problem. The Paxos Consensus algorithm [Lam96, PLL97, Lam98] considers a different system model, in which periods of synchrony and asynchrony alternate. This algorithm also proceeds in asynchronous rounds, with a coordinator for each round. Unlike in the previously mentioned algorithms, the Paxos algorithm does not use the rotating coordinator paradigm, but a perfect – in periods of synchrony – failure detector in combination with a leader election algorithm to choose the coordinator of a round. The Paxos algorithm guarantees the agreement and validity properties of consensus regardless of process, channel and timing failures and process recoveries (stable storage is used). When the system stabilizes, i.e., there are no failures nor process recoveries, and a majority of the processes are correct for a sufficiently long time, termination is guaranteed.

5.6 Solving Consensus using $\diamond\mathcal{C}$

In this section, we present a new algorithm that solves Uniform Consensus using an eventually consistent failure detector. The main difference between this algorithm and the above ones is the way the coordinator is selected. We do not use the rotating coordinator paradigm as they do, but the leader election capability of the class of failure detectors $\diamond\mathcal{C}$. In this, it has some similarities with the Paxos algorithm [Lam96, PLL97, Lam98]. However, the rest of the algorithm is much closer to the $\diamond\mathcal{S}$-Consensus algorithm of Chandra and Toueg [CT96].

5.6.1 Basic Assumptions

In the following, we assume an asynchronous system model augmented with a failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$, to which processes have access. We also assume that all processes know the function *leader* associated with $\mathcal{D}$, as specified in Definition 2. Finally, we assume that a majority of processes are correct, i.e., do not crash. Thus, if we denote by f the number of processes that can fail, we assume $f < n/2$. This is a necessary requirement to solve Consensus with an eventually consistent failure detector. This can be derived from the fact that every failure detector of class $\diamond\mathcal{P}$ is also an eventually consistent failure detector, and Theorem 6.3.1 in [CT96].

The algorithm that we present uses *Reliable Broadcast* to communicate the decision. Roughly speaking, this communication primitive allows processes to broadcast messages such that all processes agree on the set of messages they deliver, despite failures. Formally, Reliable Broadcast is a communication primitive for asynchronous systems that satisfies the following properties [HT94]:

- *R-Validity.* If a correct process broadcasts a message m, then it eventually delivers m.
- *R-Agreement.* If a correct process delivers a message m, then all correct processes eventually deliver m.
- *R-Integrity.* For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$[9].

Note that if the sender of a message m is faulty, the specification of Reliable Broadcast allows two possible outcomes: either m is delivered by all correct processes or by none.

The algorithm of figure 5.3, proposed by Chandra and Toueg in [CT96], implements Reliable Broadcast in asynchronous systems with quasi-reliable channels. Informally, when a process receives a message for the first time, it relays the message to all processes and then delivers it. This algorithm satisfies R-validity, R-agreement and R-integrity in asynchronous systems with up to $n-1$ crash failures.

Every process p executes the following:

To execute R-broadcast(m):
  send m to all (including p)

R-deliver(m) *occurs as follows*:
  when receive m for the first time
    if $sender(m) \neq p$ **then** send m to all
    R-deliver(m)

Figure 5.3: Reliable Broadcast by message diffusion.

[9] We assume that every message is unique, and that messages carry the identity of their sender.

5.6.2 The Algorithm

Figures 5.4 and 5.5 present the $\diamond\mathcal{C}$-Consensus algorithm in detail. Each process runs an instance of this algorithm, which proceeds in asynchronous rounds. As the $\diamond\mathcal{S}$-Consensus algorithm of Chandra and Toueg [CT96], it goes through three asynchronous epochs, each of which may span several rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked*: no other decision value is possible. In the third epoch, processes decide the locked value.

Each round of the algorithm is divided into five asynchronous phases. In Phase 0, every process determines its coordinator for the round. A process becomes its own coordinator for the round if it is the process returned by the function $leader$. A coordinator announces itself by sending a message to the rest of processes. A process becomes a non-coordinator, i.e., a participant, if it first receives a message from a coordinator, which becomes its coordinator for the round. In Phase 1, every process sends its current estimate of the decision value timestamped with the round number in which it adopted this estimate, to its coordinator. Also after phase 0 and concurrently with the main algorithm, each process sends a *null* estimate to the other coordinators of the round (first task of Figure 5.5).

In Phase 2, each coordinator tries to gather a majority of estimates. If it succeeds, then it selects an estimate with the largest timestamp and sends it to all the processes as a proposition. On the other hand, if it does not receive a majority of estimates then it sends a *null* proposition to all processes. In Phase 3, each process waits for a proposition from its coordinator. However, it also stops waiting if it suspects its coordinator or if it receives a non-null proposition from some other coordinator. If the process receives a non-null proposition from some coordinator (including its own), then it adopts it and sends an *ack* message to this coordinator. If the process receives a *null* proposition from its coordinator, it stops waiting and passes to the next phase. Finally, if the process suspects its coordinator, it sends a *nack* message to it. After this phase and concurrently with the main algorithm, each process sends a *nack* message to a coordinator from which it receives a non-null proposition for this round (second task of Figure 5.5). Finally, in Phase 4 the coordinator that succeeded in Phase 2 and sent a non-null proposition (if any, and as we will see at most one) waits for a majority of *ack*/*nack* messages. If it gathers a majority of *ack* messages, then it knows that a majority of processes adopted its proposition as their new estimate. Consequently, this coordinator

R-broadcasts a request to decide its proposition. At any time, if a process R-delivers such a request, it decides accordingly.

5.6.3 Proof of Correctness

In this section, we prove the correctness of the presented algorithm.

Lemma 16 *In any round r, at most one coordinator c will send a non-null estimate, i.e., a message of type (c, r, $estimate_c$), to all processes at the end of Phase 2.*

Proof: From the algorithm, it is clear that each process sends its estimate to only one coordinator in Phase 1. A coordinator must gather a majority of such estimates in order to send a non-null estimate at the end of Phase 2. Thus, at most one coordinator can gather such a majority and send a non-null estimate. ■

From the above lemma and the algorithm, it is easy to see that in any round r at most one coordinator c will R-broadcast a (c, r, $estimate_c$, *decide*) message in Phase 4[10].

Lemma 17 *No two processes decide differently.*

Proof: If no process ever decides, the lemma is trivially true. If any process decides, it must have previously R-delivered a message of type (–, –, –, *decide*). By the uniform integrity property of Reliable Broadcast and the algorithm, a coordinator previously R-broadcast this message. This coordinator must have received $\lceil (n+1)/2 \rceil$ messages of type (–, –, *ack*) in Phase 4. Let r be the smallest round number in which $\lceil (n+1)/2 \rceil$ messages of type (–, –, *ack*) are sent to a coordinator c in Phase 3. Let $estimate_c$ denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geq r$, if a coordinator c' sends $estimate_{c'}$ in Phase 2 of round r', then $estimate_{c'} = estimate_c$.

The proof is by induction on the round number. From Lemma 16 and the fact that c sends $estimate_c$ at the end of Phase 2 of round r, any other coordinator of round r sends *null_estimate* at the end of Phase 2 of round r. Thus, the claim holds for $r' = r$. Now assume that the claim holds for all r', $r \leq r' < k$. We will show that the claim holds for

[10] This is achieved through the use of the $decidible_p$ boolean variable.

Every process p executes the following:

procedure $propose(v_p)$
$estimate_p \leftarrow v_p$ $\{$*estimate*$_p$ *is p's estimate of the decision value*$\}$
$state_p \leftarrow undecided$
$r_p \leftarrow 0$ $\{r_p$ *is p's current round number*$\}$
$ts_p \leftarrow 0$ $\{ts_p$ *is the last round in which p updated estimate*$_p$*, initially 0*$\}$

while $state_p = undecided$ $\{$*Rotate until decision is reached*$\}$
$chosen_p \leftarrow false$
$replied_p \leftarrow false$
$r_p \leftarrow r_p + 1$

Phase 0: $\{$*Each process determines its coordinator for the round*$\}$
wait until $[p = leader(\mathcal{T}_p)$ **or** for a process q: received $(q, r_p, coordinator)]$ $\{$*Query the failure detector*$\}$
if [for a process q: received $(q, r_p, coordinator)$] **then**
$c_p \leftarrow q$
else
$c_p \leftarrow p$
send $(p, r_p, coordinator)$ to all processes except p
$chosen_p \leftarrow true$

Phase 1: $\{$*Each process p sends estimate*$_p$ *to its current coordinator*$\}$
send $(p, r_p, estimate_p, ts_p)$ to c_p

Phase 2: $\{$*Each coordinator tries to gather* $\lceil\frac{(n+1)}{2}\rceil$ *estimates to propose a new estimate*$\}$
if $p = c_p$ **then**
wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ **or** $(q, r_p, null\_estimate, 0)$]
$msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
if [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$] **then**
$decidible_p \leftarrow true$
$t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
$estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
send $(p, r_p, estimate_p)$ to all
else $\{$*p received null_estimate from some process*$\}$
$decidible_p \leftarrow false$
send $(p, r_p, null\_estimate)$ to all

Phase 3: $\left\{\right.$ *Each process waits for a new estimate proposed by a coordinator or to receive null_estimate from its coordinator or to suspect it* $\left.\right\}$
wait until [for a process q: received $(q, r_p, estimate_q)$ **or** received $(c_p, r_p, null\_estimate)$ from c_p **or** $c_p \in \mathcal{D}_p$]
if [for a process q: received $(q, r_p, estimate_q)$] **then** $\{$*p received estimate*$_q$ *from a process q*$\}$
$estimate_p \leftarrow estimate_q$
$ts_p \leftarrow r_p$
send (p, r_p, ack) to q
else if [received $(c_p, r_p, null\_estimate)$ from c_p] **then** $\{$*p received null_estimate from* $c_p\}$
discard message
else $\{$*p suspects that* c_p *crashed*$\}$
send $(p, r_p, nack)$ to c_p
$replied_p \leftarrow true$

Phase 4: $\left\{\right.$ *The coordinator that can still decide (if any) waits for* $\lceil\frac{(n+1)}{2}\rceil$ *replies. If they indicate that* $\lceil\frac{(n+1)}{2}\rceil$ *processes adopted its estimate, the coordinator R-broadcasts a decide message* $\left.\right\}$
if $(p = c_p)$ **and** $(decidible_p)$ **then**
wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received (q, r_p, ack) **or** $(q, r_p, nack)$]
if [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received (q, r_p, ack)] **then**
R-broadcast$(p, r_p, estimate_p, decide)$

Figure 5.4: Solving Consensus using any $\mathcal{D} \in \diamond\mathcal{C}$.

when received $(q, r_q, coordinator)$ from q such that $(r_q < r_p)$ **or** $((r_q = r_p)$ **and** $(chosen_p))$
send $(p, r_q, null\_estimate, 0)$ to q

when received $(q, r_q, estimate_q)$ from q such that $(r_q < r_p)$ **or** $((r_q = r_p)$ **and** $(replied_p))$
send $(p, r_q, nack)$ to q

{*If p R-delivers a decide message, p decides accordingly*}

when *R-deliver*$(q, r_q, estimate_q, decide)$
if $state_p = undecided$ **then**
$decide(estimate_q)$
$state_p \leftarrow decided$

Figure 5.5: Separate tasks for replying to late coordinators and taking the decision.

$r' = k$, that is, if c_k is a coordinator of round k that sends $estimate_{c_k}$ in Phase 2, then $estimate_{c_k} = estimate_c$.

From the algorithm it is clear that if c_k sends $estimate_{c_k}$ in Phase 2 of round k then it must have received estimates from at least $\lceil (n+1)/2 \rceil$ processes. Thus, there is at least one process p such that (1) p sent a (p, r, ack) message to c in Phase 3 of round r, and (2) $(p, k, estimate_p, ts_p)$ is in $msgs_{c_k}[k]$ in Phase 2 of round k. Since p sent (p, r, ack) to c in Phase 3 of round r, $ts_p = r$ at the end of Phase 3 of round r. Since ts_p is nondecreasing, $ts_p \geq r$ in Phase 1 of round k. Thus, in Phase 2 of round k, $(p, k, estimate_p, ts_p)$ is in $msgs_{c_k}[k]$ with $ts_p \geq r$. It is easy to see that there is no message $(q, k, estimate_q, ts_q)$ in $msgs_{c_k}[k]$ for which $ts_q \geq k$. Let t be the largest ts_q such that $(q, k, estimate_q, ts_q)$ is in $msgs_{c_k}[k]$. Thus, $r \leq t < k$.

In Phase 2 of round k, c_k executes $estimate_{c_k} \leftarrow estimate_q$ where $(q, k, estimate_q, t)$ is in $msgs_{c_k}[k]$ (remember that we assume that c_k sends $estimate_{c_k}$ in Phase 2 of round k). From Figure 5.4, it is clear that q adopted $estimate_q$ as its estimate in Phase 3 of round t. Thus, a coordinator of round t sent $estimate_q$ to q in Phase 2 of round t. Since $r \leq t < k$, by the induction hypothesis, $estimate_q = estimate_c$. Thus, c_k sets $estimate_{c_k} \leftarrow estimate_c$ in Phase 2 of round k. This concludes the proof of the claim.

We now show that if a process decides a value, then it decides $estimate_c$. Suppose that some process p R-delivers $(q, r_q, estimate_q, decide)$, and thus decides $estimate_q$. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast $(q, r_q, estimate_q, decide)$ in Phase 4 of round r_q. From Figure 5.4, q must have received $\lceil (n+1)/2 \rceil$ messages of type $(-, r_q, ack)$ in Phase 4 of round r_q. By the definition of r, $r \leq r_q$. From the above claim, $estimate_q = estimate_c$. ■

Lemma 18 *Every correct process eventually decides some value.*

Proof: There are two possible cases:

(1) Some correct process decides. It must have R-delivered some message of type (–, –, –, *decide*). By the agreement property of Reliable Broadcast, all correct processes eventually R-deliver this message and decide.

(2) No correct process decides. We claim that no correct process remains blocked forever at one of the **wait** statements. The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the **wait** statements. Therefore, every correct process reaches Phase 0 of round r. From the eventual consistent accuracy of $\mathcal{D}$, eventually some correct process c will become coordinator in round r, i.e., $c = leader(\mathcal{T}_c)$. Hence, c will send a message of type (c, r, *coordinator*) to the rest of processes. Thus, all correct processes reach the end of Phase 0 of round r, either by becoming coordinator or by receiving a message of type (–, r, *coordinator*) from a coordinator of round r. Thus, no correct process blocks forever at the **wait** statement of Phase 0.

It is also clear from the algorithm that all correct processes reach the end of Phase 1 of round r: they all send a message of type (–, r, *estimate*, –) to their current coordinator. Let us consider now Phase 2. For each coordinator c of round r, there are two cases to consider:

(a) From the first task of Figure 5.5 and since a majority of the processes are correct, eventually c receives at least $\lceil (n+1)/2 \rceil$ messages, either of type (–, r, *estimate*, –) or (–, r, *null_estimate*, 0), and replies by sending (c, r, $estimate_c$) or (c, r, *null_estimate*). Thus, c does not block forever at the **wait** statement in Phase 2.

(b) c crashes.

In the first case, every correct process p eventually receives (c, r, $estimate_c$) from a coordinator c (which can be its coordinator) or (c_p, r, *null_estimate*) from its coordinator c_p. In the second case, since $\mathcal{D}$ satisfies strong completeness, for every correct process p there is a time after which its coordinator c_p is permanently suspected by p, that is, $c_p \in \mathcal{D}_p$. Thus in either case, no correct process blocks at the **wait** statement in Phase 3. From Phase 3 of the algorithm and the second task of Figure 5.5, it must be clear that every correct process that receives a non-null estimate from a coordinator c, replies to it with a message of type $(-, r, ack)$ or $(-, r, nack)$. Since there are at least $\lceil (n+1)/2 \rceil$ correct processes, a coordinator that sent a non-null estimate at the end of Phase 2 will eventually receive at least $\lceil (n+1)/2 \rceil$ such messages. Note that only the coordinators that sent a non-null estimate at the end of Phase 2 execute the **wait** statement of Phase 4. Thus, no coordinator can block at the **wait** statement of Phase 4. This shows that all correct processes complete round r —a contradiction that completes the proof of our claim.

Since $\mathcal{D}$ satisfies eventual consistent accuracy, there is a correct process q and a time t such that every correct process permanently *trusts* q after t, i.e. for every correct process p: $leader(\mathcal{T}_p) = q$ after t. Let $t' > t$ be a time such that all faulty processes have crashed. Note that after t' *no* process suspects q. From this and the above claim, there must be a round r such that all correct processes reach round r after time t'. In Phase 0 of round r, q sends $(q, r, coordinator)$ to all correct processes except itself. Thus, in Phase 0 every correct process except q receives $(q, r, coordinator)$ from q and sets q as its coordinator for round r. In Phase 1, all correct processes send their estimates to q. In Phase 2, q receives $\lceil (n+1)/2 \rceil$ such estimates, and sends $(q, r, estimate_q)$ to all processes. In Phase 3, since q is not suspected by any correct process after time t', every correct process waits for q's estimate, eventually receives it, and replies with an *ack* to q. Thus, in Phase 4, q receives $\lceil (n+1)/2 \rceil$ messages of type $(-, r, ack)$ (and no message of type $(-, r, nack)$), and R-broadcasts $(q, r, estimate_q, decide)$. By the validity and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message and *decide* —a contradiction. Thus, case (2) is impossible, and this concludes the proof of the lemma. ■

Theorem 11 *The algorithm of Figures 5.4 and 5.5 solves Uniform Consensus using $\Diamond\mathcal{C}$ in asynchronous systems with $f < n/2$.*

Proof: Lemma 17 and Lemma 18 show that the algorithm of Figures 5.4 and 5.5 satisfies the uniform agreement and termination properties of Consensus, respectively. From the algorithm, it is clear that no process decides more than once, and hence the uniform integrity property holds. From the algorithm it is also clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be a value proposed by some process. Thus, uniform validity of Consensus is also satisfied. ■

5.6.4 Performance Analysis

As it was mentioned, the presented $\diamond\mathcal{C}$-Consensus algorithm does not use the rotating coordinator paradigm. Instead, the eventual leader election functionality provided by the failure detector is exploited. As a result, in the case of stability of the failure detector (i.e., the *leader* function returns the same correct process to all processes), Consensus is solved in only one round, providing early consensus. In any $\diamond\mathcal{S}$-Consensus algorithm based on the rotating coordinator paradigm, the number of rounds can be $\Omega(n)$ once the failure detector is stable (i.e., there is one correct process that is never suspected by any process), until a correct and not-suspected process becomes coordinator of a round.

With the $\diamond\mathcal{C}$-Consensus algorithm, if the failure detector is not stable, there could be multiple coordinators in the same round (until Phase 2). Then, if in Phase 0 there are $\Omega(n)$ coordinators, a round of the algorithm could have a quadratic number of messages. This happens when the *leader* function returns to $\Omega(n)$ processes their own identity. Typically, this could occur if the processes are temporarily disconnected from each other. However, in stability the number of messages sent in each phase of a round is linear. This round complexity is at least as good as that of the algorithms mentioned in Section 5.5.1.

It is worth to mention that the algorithm presented is suitable for some optimizations. These have been omitted for the sake of clarity. For instance, in Phase 2, a frustrated coordinator, i.e., a coordinator that did not receive a majority of non-null estimates, could send a *null* proposition only to those processes that sent him a non-null estimate in Phase 1.

$\infty\infty\infty$

Chapter 6

Conclusions

The *Consensus* problem is a fundamental paradigm for fault-tolerant distributed computing. However, it has been shown that it cannot be solved deterministically in an asynchronous system that is subject to even a single process crash. An elegant approach that encapsulates the minimum synchronism needed to solve Consensus in asynchronous systems is the model of *unreliable failure detectors* proposed by Chandra and Toueg. This thesis discusses the practical problem of the design of efficient algorithms that implement such failure detectors in models of partial synchrony, as well as its application to solve Consensus.

6.1 Research Assessment

This research has led to five major contributions. In the context of the implementability of failure detectors in partially synchronous systems, an impossibility result has been provided. In the context of the design of algorithms implementing failure detectors with eventual accuracy, the efficient ring-based algorithms and the optimal ordered-heartbeat algorithm have been proposed. In the context of the failure detector classification, a new class of failure detectors has been proposed. Finally, in the context of Consensus algorithms based on failure detectors, a new Consensus algorithm has been proposed.

Impossibility result. We have shown that, among the eight classes of failure detectors defined by Chandra and Toueg in [CT96], no one of the four classes satisfying perpetual accuracy can be implemented in a partially synchronous system in which bounds on message delay and processor relative speeds exist but are unknown. Intuitively, the impossibility stems from the fact that perpetual accuracy requires that the failure detector does not make any mistake on at least one correct process, which cannot be achieved if the bounds are not known.

Ring-based algorithms. We have proposed a family of algorithms that implement the four classes of failure detectors satisfying eventual accuracy. These algorithms are based on a ring arrangement of processes, and uses a polling monitoring model, having a linear message complexity.

Ordered-heartbeat algorithm. We have proposed an optimal algorithm that implements $\diamond\mathcal{S}$, the weakest failure detector for solving Consensus. This algorithm uses a heartbeat monitoring model, and satisfies the following property: eventually every correct process permanently trusts the same correct process. With the previous property, it is simple to satisfy the completeness and accuracy properties of $\diamond\mathcal{S}$: they are achieved by making each process suspect all processes except its trusted process. This algorithm performs better than previous algorithms in terms of the number of messages and the amount of information periodically exchanged. We have also proposed a new metric to evaluate the performance of algorithms implementing failure detectors, the *eventual monitoring degree*. We have shown that the ordered-heartbeat algorithm is optimal with respect to this new metric.

A new class of failure detectors. We have proposed a new class of failure detectors, the *Eventually Consistent* class, denoted $\diamond\mathcal{C}$. This class adds to the failure detection capability of other classes an eventual leader election capability. We have shown that $\diamond\mathcal{C}$ is equivalent to $\diamond\mathcal{S}$, the weakest class for solving Consensus. We have also shown that $\diamond\mathcal{C}$ can be implemented as efficiently as $\diamond\mathcal{S}$. In particular, the two algorithms implementing $\diamond\mathcal{S}$ proposed in this thesis implement also $\diamond\mathcal{C}$.

A new Consensus algorithm. We have proposed a Consensus algorithm that is based on the Eventually Consistent failure detector class. The algorithm relies strongly on the leader election capability of the class $\diamond\mathcal{C}$. To our knowledge, it is the first Consensus algorithm based on a failure detector satisfying eventual accuracy that does not rely on the rotating coordinator paradigm. The algorithm is more efficient than previously proposed algorithms in terms of the number of rounds required to reach consensus, once the failure detector is stable.

6.2 Future Directions and Open Questions

Besides the contributions presented in the previous section, this work has raised several issues that deserve further analysis. In the following, we describe some future directions and open questions related to this research.

Other applications of $\diamond\mathcal{C}$. The new class of failure detectors that we have proposed has been used to efficiently solve Consensus. A natural extension would be to investigate the use of this class – in particular its eventual leader election functionality – to solve other agreement problems, such as atomic broadcast, leader election[1], and group membership.

Relationship between the models of partial synchrony of [DDS87] and [DLS88, CT96]. In this thesis, we have proposed implementations of unreliable failure detectors in the partially synchronous model M_3 proposed by Chandra and Toueg [CT96], which is weaker than the two models proposed by Dwork et al [DLS88]. Dolev et al. have proposed in [DDS87] four minimal models of partial synchrony in which Consensus can be solved. The question that arises is if the model M_3 is equivalent to any of these four minimal models. In that case, M_3 could be viewed as the minimal model of partial synchrony in which Chandra-Toueg's failure detectors with eventual accuracy, e.g., $\diamond\mathcal{W}$, can be implemented. Furthermore, M_3 would also be a minimal model of partial synchrony in which Consensus is solvable.

What is the minimal synchronism needed for implementing $\mathcal{W}$? We have shown in this thesis that, among the eight classes of failure detectors proposed by Chandra and Toueg, no one satisfying perpetual accuracy can be implemented in the partially synchronous system M_1 in which there exist upper bounds on message delay and processor relative speeds, but the bounds are unknown. On the other hand, we know that even $\mathcal{P}$ can be implemented in a synchronous system. The question that arises is if there is some system model, necessarily

[1]Note that $\diamond\mathcal{C}$ does not solve the leader election problem, as defined in the literature. Clearly, because of the unreliability of $\diamond\mathcal{C}$, it is possible to have various leaders simultaneously.

stronger than M_1 but maybe weaker than the synchronous model, in which the weakest failure detector with perpetual accuracy, e.g, $\mathcal{W}$, can be implemented.

$\infty\infty\infty$

Bibliography

[ACT97a] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 126–140. LNCS, Springer-Verlag, September 1997.

[ACT97b] M. K. Aguilera, W. Chen, and S. Toueg. On the weakest failure detector for quiescent reliable communication. Technical report, Cornell University, Ithaca, NY, July 1997.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*, pages 231–245. LNCS, Springer-Verlag, September 1998.

[AT96] M. K. Aguilera and S. Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, pages 29–39. LNCS, Springer-Verlag, October 1996.

[ATD99] M. K. Aguilera, S. Toueg, and B. Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, pages 19–33. LNCS, Springer-Verlag, September 1999.

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th*

International Workshop on Distributed Algorithms (WDAG'96), pages 105–122. LNCS, Springer-Verlag, October 1996.

[BDM93] M. Barborak, A. Dabhura, and M. Malek. The consensus problem in distributed computing. *ACM Computing Survey*, 25(2):171–220, June 1993.

[BG93] K. P. Birman and B. B. Glade. Consistent failure reporting in reliable communication systems. Technical Report TR93-1349, Cornell University, May 1993.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Massachusetts, 1987.

[BO83] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (POC'83)*, pages 27–30, August 1983.

[Bra87] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[BT93] O. Babaoglu and S. Toueg. Non-blocking atomic commitment. In Sape J. Mullender, editor, *Distributed Systems.* ACM Press, 2nd edition, 1993.

[CD89] B. Chor and C. Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, (5):443–497, 1989.

[Cha93a] T. D. Chandra. *Unreliable Failure Detectors for Asynchronous Distributed Systems.* PhD thesis, Cornell University, New York, May 1993. Technical Report 93-1377.

[Cha93b] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 103(1):132–158, July 1993.

[Che00] W. Chen. *On the Quality of Service of Failure Detectors.* PhD thesis, Cornell University, New York, May 2000.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996. A preliminary version appeared in the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC), August 1992, 147-158.

[CHTCB96] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership in asynchronous systems. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, May 1996.

[Chu98] F. Chu. Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 67:289–293, 1998.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Cri91] F. Cristian. Understanding fault-toelarnt distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. A preliminary version titled "Unreliable failure detectors for asynchronous systems" appeared in the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC), August 1991, 325-340.

[CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (FTCS-30)*, June 2000.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–98, January 1987.

[DFKM97] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, California, August 1997.

[DGGS99] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the Third European Dependable Computing Conference (EDCC'99)*, September 1999.

[DLP+86] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DS97] A. Doudou and A. Schiper. Muteness detectors for consensus with byzantine processes. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, 1997. Presented as brief announcement in the 17th Annual ACM Symposium on Princiles of Distribute Computing (PODC'98).

[FC96] C. Fetzer and F. Cristian. Fail-aware failure detectors. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS'96)*, pages 200–209. IEEE Computer Society Press, October 1996.

[FC99] C. Fetzer and F. Cristian. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, (6):642–657, June 1999.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Proceedings of the International Conference on Foundations of Computation Theory*, volume 158, pages 127–140. LNCS, Springer-Verlag, 1983.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FM90] P. Feldman and S. Micali. An optimal algorithm for synchronous byzantine agreement. Technical Report MIT/LCS/TM-425, Massachusetts Institute of Technology, June 1990.

[GHM+00] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: a concise guided tour. In *Advances in Distributed Systems*, pages 33–47. Springer Verlag, LNCS 1752, 2000.

[GLS95] R. Guerraoui, M. Larrea, and A. Schiper. Non-blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS'95)*, pages 41–50, Bad Neuenahr, Germany, September 1995.

[GLS96] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS-16)*, pages 692–697, Hong-Kong, May 1996.

[Gro98] Object Management Group. *Fault Tolerant CORBA Using Entity Redundancy, Request For Proposal.* OMG, April 1998.

[GS96a] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe'96*, pages 38–57. Springer Verlag, LNCS 1088, June 1996. A revised version appeared in IEEE Computer 30(4), April 97, under the title "Software Based Replication for Fault-Tolerance".

[GS96b] R. Guerraoui and A. Schiper. Γ-accurate failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, pages 269–286. LNCS, Springer-Verlag, October 1996.

[GS97] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE International Workshop on Future Trends in Distributed Computing Systems (FTDCS'97)*, pages 183–188, October 1997.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95)*, pages 87–100. LNCS, Springer-Verlag, September 1995.

[HMMR00] M. Hurfin, R. Macedo, A. Mostefaoui, and M. Raynal. A sliding round window $\diamond\mathcal{S}$-based consensus protocol. Technical Report 1304, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, February 2000.

[HMR97] M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. Technical Report 1144, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, November 1997.

[HMR00] M. Hurfin, A. Mostefaoui, and M. Raynal. A comprehensive approach for failure detector-based consensus protocols. Technical Report 1303, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, February 2000.

[HR99] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Addison-Wesley, 2nd edition, 1993.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Ithaca, NY, 1994.

[LAF99] M. Larrea, S. Arévalo, and A. Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on DIstributed Computing (DISC'99)*, pages 34–48. LNCS, Springer-Verlag, September 1999.

[Lam96] B. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, pages 1–15. LNCS, Springer-Verlag, October 1996.

[Lam98] L. Lamport. The part-time parliament. *ACM Transsctions on Computer Systems*, 16(2):133–169, May 1998.

[LFA00a] M. Larrea, A. Fernández, and S. Arévalo. Brief announcement: Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'2000)*, Portland, Oregon, July 2000. To appear.

[LFA00b] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. Technical Report, Universidad Pública de Navarra, April 2000. Accepted as brief announcement in the 14th International Symposium on Distributed Computing (DISC'2000), Toledo, Spain.

[LFA00c] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, Nurenberg, Germany, October 2000. To appear.

[LO88] B. H. Liskov and B. M. Oki. Viewstamped replication: A new primary copy method to support high-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, August 1988.

[Mal96] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1996.

[MR99a] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. In *Proceedings of the 13th International Symposium on DIstributed Computing (DISC'99)*, pages 49–63. LNCS, Springer-Verlag, September 1999.

[MR99b] A. Mostefaoui and M. Raynal. Unreliable failure detectors with limited scope accuracy and an application to consensus. In *Proceedings of the 19th International Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS'99*, pages 329–340. LNCS, Springer-Verlag, December 1999.

[MRF99] A. Mostefaoui, M. Raynal, and F.Tronel. The best of both worlds: a hybrid approach to solve consensus. Technical Report 1280, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, November 1999.

[NT90] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[OGS97] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report TR97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.

[PLL97] R. De Prisco, B. Lampson, and N. A. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 111–125. LNCS, Springer-Verlag, September 1997.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Rab83] M. Rabin. Randomized Byzantine Generals. In *Proceedings of the 24th Annual ACM Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[RB91] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 341–351, August 1991.

[RT99a] M. Raynal and F. Tronel. Group membership failure detection: A simple protocol and its probabilistic analysis. *Distributed Systems Engineering Journal*, 6(3):95–102, 1999.

[RT99b] M. Raynal and F. Tronel. Restricted failure detectors: Definition and reduction protocols. *Information Precessing Letters*, 72(3-4):91–97, November 1999.

[Sab95] L. S. Sabel. Approximating perfect failure detectors in asynchronous distributed systems. Technical Report TR95-1550, Cornell University, October 1995.

[Sch90] F. B. Schneider. Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[SDS99] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[Ser98] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[Ske81] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–142, 1981.

[SM95] L. S. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Ithaca, NY, February 1995.

[SR93] A. Schiper and A. M. Ricciardi. Virtually-synchronous communication based on weak failure suspectors. In *Proceeding of the IEEE 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 534–542, June 1993.

[Tou84] S. Toueg. Randomized byzantine agreement. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 163–178, August 1984.

[TS92] J. Turek and D. Shasha. The many faces of consensus in distribued systems. *IEEE Computer*, 25(6):8–17, June 1992.

[vRMH98] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIF International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[YNG98] J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 297–308, 1998.

[Zam96] A. Zamsky. A randomized byzantine agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, May 1996.

Appendix A

From Weak Completeness to Strong Completeness

Every process p executes the following:

$output_p \leftarrow \emptyset$

cobegin
|| *Task 1:* **repeat forever**
  $\{p$ *queries its local failure detector module* $\mathcal{D}_p\}$
  $suspects_p \leftarrow \mathcal{D}_p$
  send $(p,\ suspects_p)$ to all

|| *Task 2:* **when** receive $(q,\ suspects_q)$ for some q
  $output_p \leftarrow (output_p \cup suspects_q) - \{q\}$ $\{output_p$ *provides strong completeness*$\}$
coend

Figure A.1: From Weak Completeness to Strong Completeness.

$\infty\infty\infty$

Appendix B

Chandra-Toueg Consensus Algorithm for $\diamond\mathcal{S}$

$\infty\infty\infty$

Every process p executes the following:

procedure $propose(v_p)$
$estimate_p \leftarrow v_p$ $\{estimate_p$ *is p's estimate of the decision value*$\}$
$state_p \leftarrow undecided$
$r_p \leftarrow 0$ $\{r_p$ *is p's current round number*$\}$
$ts_p \leftarrow 0$ $\{ts_p$ *is the last round in which p updated* $estimate_p$*, initially 0*$\}$

$\{$*Rotate through coordinators until decision is reached*$\}$

while $state_p = undecided$
$r_p \leftarrow r_p + 1$
$c_p \leftarrow (r_p \bmod n) + 1$ $\{c_p$ *is the current coordinator*$\}$

Phase 1: $\{$*All processes p send* $estimate_p$ *to the current coordinator*$\}$
send $(p, r_p, estimate_p, ts_p)$ to c_p

Phase 2: $\{$*The current coordinator gathers* $\lceil\frac{(n+1)}{2}\rceil$ *estimates and proposes a new estimate*$\}$
if $p = c_p$ **then**
wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ from q]
$msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
$t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
$estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
send $(p, r_p, estimate_p)$ to all

Phase 3: $\{$*All processes wait for the new estimate proposed by the current coordinator*$\}$
wait until [received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $c_p \in \mathcal{D}_p$] $\{$*Query the failure detector*$\}$
if [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then** $\{p$ *received* $estimate_{c_p}$ *from* $c_p\}$
$estimate_p \leftarrow estimate_{c_p}$
$ts_p \leftarrow r_p$
send (p, r_p, ack) to c_p
else $\{p$ *suspects that* c_p *crashed*$\}$
send $(p, r_p, nack)$ to c_p

Phase 4: $\left\{\begin{array}{l}\textit{The current coordinator waits for } \lceil\frac{(n+1)}{2}\rceil \textit{ replies. If they indicate that } \lceil\frac{(n+1)}{2}\rceil \\ \textit{processes adopted its estimate, the coordinator R-broadcasts a decide message}\end{array}\right\}$
if $(p = c_p)$ **then**
wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received (q, r_p, ack) **or** $(q, r_p, nack)$]
if [for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received (q, r_p, ack)] **then**
R-broadcast$(p, r_p, estimate_p, decide)$

$\{$*If p R-delivers a decide message, p decides accordingly*$\}$

when *R-deliver*$(q, r_q, estimate_q, decide)$
if $state_p = undecided$ **then**
$decide(estimate_q)$
$state_p \leftarrow decided$

Figure B.1: Chandra-Toueg Consensus Algorithm for $\Diamond\mathcal{S}$.

Curriculum Vitae

Mikel Larrea was born on March 25, 1970, in Oñati, Spain. He attended primary and secondary school in Oñati, obtaining his high school diploma in 1988. In 1992, he graduated as an electrical engineer from Mondragon Eskola Politeknikoa, a polytechnical school in Mondragón (Spain). During this period, from 1990 to 1992, he also worked on projects for the Ikerlan Research Center in Mondragón. From 1992 to 1995, he studied Computer Science at the Swiss Federal Institute of Technology in Lausanne (EPFL, Switzerland), graduating in March 1995. During 1995, he worked on research in distributed systems at the Operating Systems and Software Engineering laboratories at EPFL. In 1996, he returned to Spain, starting his Ph.D. under the supervision of Professor André Schiper (EPFL, Switzerland) and Dr. Sergio Arévalo (University Rey Juan Carlos, Spain). From 1996 to 1998, he worked as a research and teaching assistant at the Mondragon High Polytechnic School. In November 1998, he joined the Distributed Systems Group at the Public University of Navarra.