Universidad del País Vasco
Euskal Herriko Unibertsitatea

THE OMEGA FAILURE DETECTOR IN THE CRASH-RECOVERY MODEL

Dissertation for the degree of
Doctor of Philosophy presented by

Cristian Martín

Advisor

Mikel Larrea

Computer Architecture and Technology department
Computer Science Faculty
UNIVERSITY OF THE BASQUE COUNTRY

November, 2010

ISBN: 978-84-694-1126-1

”The aim of science is not to open the door to infinite wisdom,
but to set a limit to infinite error.”
Bertolt Brecht

”A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.”
Albert Einstein

Acknowledgements

I am grateful to many people for help, both direct and indirect, in writing this thesis. First, I would like to thank Dr. Mikel Larrea, my advisor and friend. Not only he always *trusted* me, but he is a great *model* in the area of distributed systems. I would like to express my gratitude to the members of the Distributed Systems Group at the University of the Basque Country, and especially to Roberto Cortiñas, for helping me in many ways. Also, I would like to thank IKERLAN its support in the later stages of this thesis.

En un plano más personal, me gustaría agradecer a mis padres su cariño y apoyo, y en particular sus esfuerzos en todo lo referente a mi educación. Los cuentos, las multiplicaciones, las figuras de papel, el ordenador MSX, el piso de estudiantes... ¡Os agradezco tantas cosas!

Y cómo no, me gustaría agradecer a Pakene, mi flor, toda la comprensión y la paciencia que me demuestra cada día en las cosas importantes.

Abstract

The design and verification of fault-tolerant distributed algorithms and applications are complex tasks. In order to study them, several standard problems have been identified. One of the most important is *Consensus*, the problem of a number of processes trying to agree on a common decision. The Consensus problem can not be solved deterministically in asynchronous systems where processes can crash. In order to circumvent this impossibility Chandra and Toueg proposed the *unreliable failure detectors*.

In this dissertation we study, for the first time, the *Omega*, Ω, (unreliable) failure detector in the *Crash-Recovery System Model*. More specifically we focus on the design of algorithms that implement this failure detector in models of partial synchrony where processes can crash and later recover, in which Consensus has been proven to be solvable.

We first redefine the Omega failure detector for the crash-recovery model. We define the $Omega_{cr1}$ and $Omega_{cr2}$ failure detectors for systems without and with stable storage respectively. Then, we propose a set of eight distributed algorithms that work in (slightly) different crash-recovery system models. With regard to efficiency, we have implemented two communication-efficient algorithms, one for $Omega_{cr1}$ and another one for $Omega_{cr2}$.

Additionally, we propose two algorithms implementing adaptations of the Eventually Perfect, $\Diamond\mathcal{P}$, failure detector. In the crash-recovery model, it is not possible to implement a failure detector of the class $\Diamond\mathcal{P}$. For this reason, we have defined and

implemented the $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$ failure detectors, which satisfy weaker properties. The algorithms rely heavily on the use of the leader election service provided by the $Omega_{cr2}$ failure detector. Finally, we propose three aggregator election and data aggregation algorithms for wireless sensor networks built on top of our implementations of the $Omega_{cr2}$ failure detector.

Resumen

El diseño y la verificación de algoritmos y aplicaciones distribuidas tolerantes a fallos son tareas complejas. Para estudiarlas, se han identificado varios problemas estándar. Uno de los más importantes es el *Consenso*, el problema de varios procesos intentando acordar una decisión común. El problema del Consenso no puede ser resuelto determinísticamente en sistemas asíncronos donde los procesos pueden fallar. Para salvar esta imposibilidad, Chandra y Toueg propusieron los *detectores no fiables de fallos*.

En esta tesis estudiamos, por primera vez, el detector no fiable de fallos *Omega*, Ω, en el *modelo de sistema de fallo-y-recuperación* (Crash-Recovery). Más concretamente nos centramos en el diseño de algoritmos que implementan dicho detector de fallos en modelos de sincronía parcial donde los procesos pueden caer y luego recuperarse, para los que se ha demostrado que el Consenso se puede resolver.

En primer lugar redefinimos el detector de fallos Omega para el modelo de fallo-y-recuperación. Definimos los detectores de fallos $Omega_{cr1}$ y $Omega_{cr2}$ para sistemas sin y con memoria estable respectivamente. Seguidamente, proponemos un conjunto de ocho algoritmos distribuidos que funcionan en modelos de sistema de fallo-y-recuperación (ligeramente) diferentes. Respecto a la eficiencia, se han implementado dos algoritmos eficientes en cuanto a comunicación (communication-efficient), uno para $Omega_{cr1}$ y el otro para $Omega_{cr2}$.

Además, proponemos dos algoritmos que implementan detectores de fallos *Eventually Perfect*, $\diamond\mathcal{P}$. En el modelo de fallo-y-recuperación, no es posible implementar un

detector de fallos de la clase $\diamond\mathcal{P}$. Por ello, se han definido e implementado los detectores de fallos $\diamond\mathcal{P}_{cr}$ y $\diamond\mathcal{P}_{k-cr}$, que satisfacen propiedades más débiles. Los algoritmos están basados en el uso de un servicio de elección de líder, que es proporcionado por el detector de fallos $Omega_{cr2}$. Finalmente, proponemos tres algoritmos de elección de *agregador* y *agregación de datos* para redes de sensores inalámbricas, construidos sobre nuestras implementaciones del detector de fallos $Omega_{cr2}$.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Background

This dissertation is contextualized in the field of distributed systems and hence we will start with a set of common definitions of this domain.

First of all we should know what a distributed system is. According to the free on-line dictionary of computing [1] a distributed system is *"A collection of (probably heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent"*.

The term *automata* used in this formal definition refers to very diverse types of devices such as computers, mobile phones, sensors... A typical example of a distributed system is an automated banking system. A user can access an automated teller machine (ATM) where transactions can be made, and every transaction must be replicated on several servers in different locations. Furthermore, different users could access the same bank account at the same time from different ATMs. However, a user only deals with a local ATM and she or he will probably never be aware of the distributed system beyond the ATM.

In this dissertation we will repeatedly use the term *process*. We will call process to a processor which is able to execute an algorithm. We do not care about the hardware and software behind the process.

In relation to distributed systems we have the field of distributed computing. This deals with distributed hardware and software systems that, emphasize the transparency of the distribution, so that the user of an application at a given automaton (e.g. a computer) is unable to distinguish between the local and the remote parts of the distributed system. For this purpose, the software required for the distributed hardware deliberately hides the distributed nature of the system.

In a distributed system, the logic of the algorithm that solves a particular problem

must often be distributed among the components of the system. We will call an algorithm designed to run on a distributed system a *distributed algorithm*. This dissertation focuses on the design of distributed algorithms in the field of unreliable failure detectors, which will be introduced later.

Nancy Lynch, one of the gurus in the area of distributed algorithms, wrote in her seminal book [94]:

"A distributed algorithm is an algorithm designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in various application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing, and real-time process control. Standard problems solved by distributed algorithms include leader election, consensus, distributed search, spanning tree generation, mutual exclusion, and resource allocation.

Distributed algorithms are typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing. One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behaviour of the independent parts of the algorithm in the face of processor failures and unreliable communications links. The choice of an appropriate distributed algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system the algorithm will run on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes."

Although nowadays we should speak about intelligent or processing devices rather than computer hardware, this definition, given in 1996, captures the essence.

Distributed systems can be modelled or defined making different assumptions about system behaviour and/or features. We will call a distributed system with its particular assumptions a *distributed system model*, *system model* or just *system*. For instance, in

[45] Dolev et al. considered a distributed system composed of interconnected processors with five key assumptions or parameters, each of which may or not be satisfied, making a total of $2^5 = 32$ system models.

One of the most important assumptions in which distributed system models can differ is related to timing aspects. In fact, most models focus on two timing attributes: the time needed for a message sent by one process to reach its destination and be delivered by another process; and the time taken by a process to execute a part of the algorithm.

We can classify these attributes as synchronous, asynchronous or partially synchronous, depending on the timing assumptions made. A timing attribute is synchronous if there is a *known* fixed upper *bound* on it; it is asynchronous if there is *no bound* on it; and if it is neither synchronous nor asynchronous it is referred to as partially synchronous.

With respect to the system models defined in terms of these attributes: a distributed system model is considered asynchronous if at least one of the previous timing attributes is asynchronous and is considered synchronous if both attributes are synchronous. Finally, if it is neither synchronous nor asynchronous then the distributed system model is partially synchronous.

Another important assumption for a distributed system model concerns the *failure types* that can happen in it. Some well known process failure types are:

- *Crash*. Also known as *halting failure* and *crash-stop*. The process simply stops forever. There is no way to detect the failure except by time out; it either stops sending messages or fails to respond to requests.
- *Crash-recovery*. The process stops and later recovers.
- *Fail-stop*. A crash failure with some kind of notification to other processes.
- *Omission*. Failure to send and/or receive messages. They can be permanent or transient.

- *Byzantine*. This failure type encompasses several types of faulty behaviours including data corruption or loss, failures caused by malicious programs, etc.

1.1.1 Consensus

In distributed systems, the design and verification of fault-tolerant (distributed) algorithms and applications are considered complex tasks. In order to study the aforementioned tasks, several standard problems have been identified. One of the most important is the *Consensus* problem [57, 110]. Basically, Consensus algorithms solve the problem of a number of processes trying to agree on a common decision. More precisely, a Consensus protocol must satisfy the four formal properties below:

(1) *Termination*: Every correct process decides some value.

(2) *Uniform Integrity*: Every process decides at most once.

(3) *Agreement*: No two correct processes decide differently.

(4) *Validity*: If a process decides *v*, then *v* was proposed by some process.

The Consensus problem has been studied extensively in different distributed system models. The importance of Consensus relies on the fact that other agreement problems such as atomic commitment [16, 19, 30, 49, 67, 68], group membership [11, 22, 27, 64, 72, 97, 100, 111, 118], and totally ordered broadcast (also known as *atomic* broadcast) [22, 28, 29, 40, 41, 51, 52, 71, 119] can be reduced to some form of Consensus, and hence solutions to these problems can be built on top of a Consensus algorithm.

It is well known that Consensus can be easily solved in a synchronous distributed system [53, 127] because it is generally easy to reliably detect the process crashes using time-outs. In an asynchronous distributed system, Fischer et at. showed in [59] that Consensus cannot be solved deterministically if at least one process can fail, even by crashing. This is known as the *FLP impossibility* result. Consequently, many problems

that require all processes to participate actively in the resolution are not deterministically solvable in an asynchronous system [27, 59, 65, 115, 120].

Basically, the FLP impossibility is due to the fact that in asynchronous systems it is not possible to distinguish a faulty process from a very slow one. This implies that in such systems reliable failure detection is not possible.

Observe that although the asynchronous model is attractive, the impossibility of deterministic solutions to the Consensus problem is a major drawback when designing fault-tolerant algorithms.

With regard to Consensus solvability [58], we can find several works about lower bounds on what Consensus algorithms can achieve. In [86] Lamport reviews previous works [31, 81] and presents new results. In summary, Lamport presents two results: a majority of processes is required to be correct to ensure progress; and if more than one process is allowed to propose, it is required that more than two-thirds of the processes are correct to reach a decision in two communication steps. These bounds show a trade-off between resilience and the number of communication steps needed to solve Consensus.

In order to circumvent the FLP impossibility result while solving Consensus, several lines of research emerged, including the following:

- the definition of weaker problems than Consensus and their solution, as in [14, 20, 21, 47];
- the use of *randomization* techniques, as in [8, 13, 18, 37, 113, 121]; and
- the study of partially synchronous models, as in [15, 45, 50].

When using randomization techniques, the randomness can be introduced to the model in two ways. On the one hand, the model itself can be randomized. This means that in a given state concrete operations only occur with some probability [12, 23]. This approach is not very popular due to its limited practical applicability.

On the other hand, there is a more realistic approach based on *randomized algorithms* that employ a degree of randomness as a part of their logic. Processes have access to operations that return random values according to specified probability distributions that are used as additional inputs for different purposes. These operations, often called *coin-flips*, are enough to overcome some impossibility results with an average probability of 1, and can also be used to achieve good performance in the average case.

Deterministic algorithms, unlike randomized algorithms, always behave predictably: i.e. given a particular input, they always produce the same output. However, this behaviour can cause a worse performance or the need to augment the system in order to solve a particular problem.

The strategy followed in the study of partially synchronous systems is to enhance the asynchronous system to some extent in order to make Consensus, or any other distributed problem, solvable. In this way, some works have focused on the identification of the minimum amount of synchrony required to solve Consensus in a distributed system where processes can crash. The works discussed below consider distributed system models composed of a finite set of processes that communicate by exchanging messages.

Dolev et al. in [45] delineate the boundary between solvability and unsolvability of Consensus in a specific area of system models. They defined 32 different models and identified four *minimal* models in which Consensus is solvable. Roughly speaking, they considered five critical parameters, that can be set to either *favourable* "F" or *unfavourable* "U". The 32 models correspond to particular settings of the five parameters, i.e. $2^5 = 32$. The critical parameters are:

(1) processors are synchronous "F" or asynchronous "U",

(2) communication is synchronous "F" or asynchronous "U",

(3) message order is synchronous "F" or asynchronous "U",

(4) broadcast transmission ”F” or point-to-point transmission ”U”, and

(5) atomic receive/send ”F” or separate receive and send ”U”.

For example, one minimal instance that allows solving Consensus is a system model where parameters (1) and (2) are favourable; i.e. there are synchronous processors and synchronous communication. It is irrelevant whether the rest of the parameters are unfavourable or favourable, because the solvability of the Consensus problem in the distributed system model is guaranteed by the first two favourable parameters.

Dwork et al. [50] considered two models of partial synchrony. Basically, the first model, referred to as *M1* in some works, stipulates that in every execution there are bounds on message transmission times and on relative process speeds, but these bounds are not known a priori. The second model, *M2*, considers that these bounds are known but they hold only after some unknown, but finite, time. In their work they showed that Consensus can be solved in both models, assuming a majority of correct processes.

In [28] Chandra and Toueg presented the partially synchronous system model *M3*, composed of a set of processes that communicate by sending messages through communication links. The partially synchronous system model *M3* defines a new kind of partial synchrony in which the timing attributes are bounded but the bounds are unknown and only hold after an unknown but finite global stabilization time. In other words, *M3* combines the characteristics of *M1* and *M2*.

More recently, Widder et al. presented the Theta-Model [125], which is a partially synchronous model that allows synchrony to be achieved without clocks. In brief, the asynchronous system is enhanced with a bound Θ on the ratio of the longest and shortest end-to-end delays of messages that are in transit simultaneously. The bound Θ can be unknown and can hold only after an unknown time that, as in other works, is called the global stabilization time. They also prove that in this model it is possible to solve Consensus and other distributed problems, even with Byzantine failures.

In [32] a new and disruptive model is proposed: the *Heard-of* model, denoted by *HO*. According to the authors, the *HO* model is inspired by [50, 62, 116]. Charron-Bost and Schiper question three general assumptions in fault-tolerant distributed systems. They consider that:

- The degree of synchrony and the failure model for processes and links should not be independent parameters.
- Distinguishing between process failures and link failures should be avoided.
- The definition of Consensus should not refer to the process fault model.

Therefore, the *HO* model encapsulates the degree of synchrony and the fault model in the same structure, and the distinction between process and link failures disappears. The computation consists of asynchronous communication-closed rounds. In every round each process sends a message to the rest of the processes and receives messages from the other processes. If a message is not received in a round it is lost. At a process p in a round r the set of processes from which p has received a message is called the heard-of set, denoted by $HO(p,r)$. Finally, they analyze previous results and Consensus algorithms in their model.

1.1.2 Failure Detectors

As already mentioned, the solvability of some distributed problems, such as the Consensus problem, depends largely on the possibility of detecting process failures. For instance, in a synchronous system, a process can reliably detect if another process has failed by the use of time-outs: a process can set one local timer to a known time-out value, the upper bound on message delivery plus the maximum processing time. If the timer expires, i.e. the time ends before the reception of the message, it means that the other process has failed (assuming that communication is reliable). Hence, a reliable detection of failures is possible and the Consensus problem can be solved. In an

asynchronous system such a detection is not possible, making Consensus unsolvable. However, as stated in the previous section a fully synchronous system is not mandatory for solving Consensus and different solutions are given in order to circumvent the FLP impossibility result.

Chandra and Toueg in [28] proposed an apparently alternative approach to overcoming the FLP impossibility result by augmenting the asynchronous system with an *unreliable failure detector*. Roughly speaking, an unreliable failure detector is a service, often called an *oracle*, that provides information about the (crash) failures of the processes that compose a system and it is implemented by a distributed algorithm. This information can be erroneous at a given time, but the behaviour of the algorithm must satisfy two properties. Depending on the properties that an unreliable failure detector satisfies, it belongs to one or another of the eight different *classes* of unreliable failure detectors they defined. They showed that Consensus can be solved in an asynchronous system provided with any of the failure detectors they proposed.

Although at first sight unreliable failure detectors can seem a different line of research from the study of partially synchronous systems, we consider they are not. The failure detectors are defined in terms of two abstract properties called *completeness* and *accuracy*, but these properties cannot be implemented in an asynchronous system, as the FLP impossibility states it is not possible to carry out any kind of reliable (crash) failure detection in a pure asynchronous system. Hence, the synchrony assumptions are encapsulated inside the unreliable failure detector, possibly eliminating any reference to *time* while solving Consensus. This line of research can be viewed as an abstract way of incorporating partial synchrony assumptions in the system.

When dealing with the problem of implementing any class of unreliable failure detector, we will have to make some synchrony assumptions. Therefore, although theoretically we can solve Consensus in an asynchronous system with an unreliable failure detector, the system needs to be partially synchronous in order to implement the failure

detector. Note that if we could solve Consensus deterministically in an asynchronous system with the help of an unreliable failure detector that can also be implemented in an asynchronous system, we would be contradicting the FLP impossibility result. On the other hand, in a fully synchronous system we can easily build *reliable* or unreliable failure detectors or even solve Consensus directly, without the help of an unreliable failure detector.

In any case the use of unreliable failure detectors not only allows the Consensus problem to be solved by providing implemented failure detectors to the Consensus algorithm, but allows the study of the Consensus problem in pure asynchronous systems by avoiding the references to time while solving it.

We will now present the unreliable failure detectors proposed by Chandra and Toueg more formally. For the exact definition of the unreliable failure detectors please read the seminal work [28]. They considered distributed failure detectors where each process has access to a local failure detector module. Basically, the module maintains a local list of processes that it currently suspects to have crashed. This list can change through time and the failure detector module can even suspect processes that have not crashed and include them in the list. When the module realizes that a process was erroneously suspected, it just removes the process from the list. Thus, a process can be repeatedly included and removed from a list. Different processes can suspect differently at a given time and hence maintain distinct lists of suspected processes. In fact, a process that has crashed may never be suspected, or a process that never crashes may be repeatedly suspected. These erroneous suspicions correspond to a normal behaviour of an unreliable failure detector.

Unreliable failure detectors are defined in terms of two abstract properties. Chandra and Toueg present the implementation of an unreliable failure detector in a partially synchronous system in [28], but just to demonstrate that implementing such an abstraction is possible. The properties that characterize an unreliable failure detector *class* are

completeness and *accuracy*. Basically, completeness specifies the capacity of a failure detector to suspect crashed processes and accuracy limits the mistakes of the failure detector while suspecting. They present eight classes of failure detectors combining two completeness and four accuracy properties.

The two completeness properties are:

(C.1) *Strong completeness*. Eventually every process that crashes is permanently suspected by every correct process.

(C.2) *Weak completeness*. Eventually every process that crashes is permanently suspected by some correct process.

As the completeness properties are easily implementable by permanently suspecting the other processes in the system, the accuracy property restricts the mistakes while suspecting.

The four accuracy properties are:

(A.1) *Strong accuracy*. No process is suspected before it crashes.

(A.2) *Weak accuracy*. Some correct process is never suspected.

(A.3) *Eventual strong accuracy*. There is a time after which correct processes are not suspected by any correct process.

(A.4) *Eventual weak accuracy*. There is a time after which some correct process is never suspected by any correct process.

In a real distributed system it is very difficult to implement strong accuracy and weak accuracy properties, called *perpetual accuracy* properties, due to the mandatory absence of mistakes from the beginning of the algorithm execution (at least on some correct process in the case of the weak accuracy property). In fact, Larrea et al. in [89] showed that it is not possible to implement them in the classical partially synchronous

systems *M1*, *M2* or *M3* where processes can crash. The implementation of the *eventual accuracy* properties is more natural, although failure detectors of the eight classes have been studied.

Chandra and Toueg establish a hierarchy of failure detector classes. Roughly speaking, a class includes the set of failure detectors that capture the same information about process failures. In the hierarchy, if the information about process failures provided by a failure detector of the class *A* includes the information provided by a failure detector of the class *B*, we say that class *A* is stronger than class *B*. If we focus on the eventual weak accuracy property, we can see from the definition that there is a time after which at least ***one*** correct process is never suspected by any correct process, while in the eventual strong accuracy property ***all*** the correct processes must eventually not be suspected. Clearly, a failure detector that satisfies eventual strong accuracy also satisfies eventual weak accuracy.

In Table 1.1 we can find the eight classes resulting from the combination of the completeness and accuracy properties proposed by Chandra and Toueg in [28].

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	$\mathcal{Q}$	$\mathcal{W}$	$\diamond\mathcal{Q}$	$\diamond\mathcal{W}$

Table 1.1: Classes of unreliable failure detectors.

In their work, Chandra and Toueg stated that any unreliable failure detector that satisfies weak completeness can be transformed into a failure detector that satisfies strong completeness with the same accuracy property. Thus, the study of the failure detectors was able to focus on those that satisfy the strong completeness property, which are shown in the first row of Table 1.1.

However, they also demonstrated that the Consensus problem can be solved with a failure detector from any of the eight classes in a system with a majority of correct

processes. Hence the weakest failure detector that allows solving Consensus is the eventually weak $\diamond\mathcal{W}$ satisfying that (1) eventually every process that crashes is permanently suspected by some correct process, and (2) there is a time after which some correct process is never suspected by any correct process.

With regard to this, in [26] Chandra et al. study the $\diamond\mathcal{W}$ failure detector in-depth, showing that it is the weakest failure detector for solving Consensus in an asynchronous system with a majority of correct processes. In other words, any failure detector that allows solving Consensus is at least as strong as the $\diamond\mathcal{W}$ failure detector. In order to prove this, they defined a new failure detector, *Omega*, also denoted as Ω in the literature, which is equivalent to $\diamond\mathcal{W}$. The output of an Omega module at a process p is the identity of a single process q that p considers to be correct. We can also say that the process q is *trusted* by p. More formally, the Omega failure detector satisfies the following property [26]:

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

The property can be seen as a *leader election* because, in brief, it states that the correct processes eventually agree on the correctness of one process, the trusted process, which can be considered an elected *leader*.

When Omega was defined for the first time it was referred to as the *failure detector Omega* (Ω), and it has been referred to in this way in most of the works about it. We have maintained this expression in this dissertation, although we consider it would be more accurate to refer to it as the *failure detector class Omega*. In any case, we will use *failure detector (class) Omega* to refer to the set of failure detectors that satisfy the property indicated above. For easy of reading, we will often use *algorithm implementing Omega*, or *Omega algorithm* to refer to an algorithm that implements a failure detector satisfying the property defined for Omega; i.e. an implementation of a failure detector in the Omega class. For the same reason, we will sometimes use Omega in the wider

sense of the term; i.e. to refer to the set of failure detectors that satisfy the properties defined for *Omega*, $Omega_{cr1}$ and $Omega_{cr2}$, which will be presented later.

In order to prove the equivalence between Omega and $\diamond\mathcal{W}$ we can describe the transformation from Omega to $\diamond\mathcal{W}$ and vice versa. The transformation of Omega into $\diamond\mathcal{W}$ is intuitive. A process that has access to an Omega module only has to suspect all the processes except the process trusted by the Omega module in order to obtain the list of suspected processes that satisfies the weak completeness and eventual weak accuracy properties of $\diamond\mathcal{W}$. The transformation in the other direction, i.e. from $\diamond\mathcal{W}$ to Omega, is more complicated and can be found in [26, 38].

1.2 Motivation

At this point it would be appropriate to recall that the title of this dissertation is *The Omega Failure Detector in the Crash-Recovery Model*. The type of failure considered when Omega was defined was the crash failure. In this dissertation we study, for the first time, the Omega failure detector in the *crash-recovery* failure model.

Although the crash failure model is of great theoretical interest, the crash-recovery failure model allows us to model more practical scenarios. For example, a real scenario in which a process crashes and after a reboot of the computer the process continues from a recovery point.

The study of Omega itself is significant because:

(1) Omega provides a *leader election* service; and

(2) it can be used for solving Consensus.

In this regard, the interest in Omega is shown by the number of classical and recent works that can be found about algorithms implementing Omega and algorithms using Omega to solve Consensus and other problems. Some of these will be reviewed in Chapter 2.

Furthermore, although not studied in this dissertation, algorithms proposed for the crash-recovery model can be adapted to dynamic distributed systems where processes can intermittently join and leave the system.

1.3 Summary of Contributions

The main goal of this dissertation is to study the Omega failure detector in a crash-recovery system model. More specifically, we focus on the design of algorithms that implement this failure detector in models of partial synchrony subject to crash-recovery failures. As a result, this research provides four major contributions.

- **The redefinition of the Omega failure detector for the crash-recovery model**. The definition of Omega is well suited to the crash model, but it can be improved in the crash-recovery model. The definition of Omega does not take into account unstable processes, i.e. processes that crash and recover infinitely often, and hence they are allowed to permanently disagree with correct processes, which can be a serious drawback. For this reason we have defined the $Omega_{cr1}$ and $Omega_{cr2}$ failure detectors. Basically, the $Omega_{cr2}$ failure detector establishes that correct processes and unstable processes, when up, will permanently agree on the same correct leader. With the $Omega_{cr1}$ failure detector, unstable processes do not trust any process upon recovery, and if they trust a process it will be the correct leader. As we will see, the $Omega_{cr2}$ failure detector requires a system where processes have access to stable storage (or some equivalent mechanism) while $Omega_{cr1}$ does not.

- **A collection of algorithms that implement *Omega*, $Omega_{cr1}$ or $Omega_{cr2}$.** Our main contribution is a set of eight distributed algorithms that work in (slightly) different system models where processes are subject to crash-recovery failures. In

this context, we have reflected on the limits of the synchrony required to implement *Omega*$_{cr1}$ and *Omega*$_{cr2}$. With regard to efficiency, we have implemented two communication-efficient algorithms: one for *Omega*$_{cr1}$ in a system without stable storage, based on nondecreasing local clocks; and another for *Omega*$_{cr2}$, where processes have access to stable storage.

- **Two algorithms implementing eventually perfect failure detectors**. In the proposed distributed systems, subject to crash-recovery failures, it is not possible to implement a failure detector of the class $\diamond\mathcal{P}$. Basically, in such a system we cannot distinguish an unstable process from an eventually up (correct) process that has not yet stabilized. For this reason, we have defined the $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$ failure detectors which satisfy weaker properties but which are achievable in the crash-recovery model. In addition, we have presented two algorithms implementing $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$. The algorithms rely strongly on the use of the leader election service provided by the *Omega*$_{cr2}$ failure detector.

- **Three aggregator election and data aggregation algorithms for wireless sensor networks**. A wireless sensor network, WSN, can be seen as a distributed system subject to crash-recovery failures. On this basis, we have built three hierarchical aggregator election and data aggregation algorithms for large WSNs on top of our implementations of the *Omega*$_{cr2}$ failure detector.

Table 1.2 shows the system models and the algorithms proposed in this dissertation implementing the *Omega*, *Omega*$_{cr1}$ or *Omega*$_{cr2}$ failure detectors.

1.4 Thesis Outline

In this chapter we have given an introduction to the topic of this dissertation.

In Chapter 2 we review the state of the art regarding the Omega failure detector and

System Model	Communication Efficient	Stable Storage	Omega Property	Known Membership
S_1	No	No	$Omega_{cr1}$	Yes
S_2	No	Yes	$Omega_{cr2}$	No
S_3	No	Yes	$Omega_{cr2}$	Yes
S_4	No	Yes	$Omega_{cr2}$	No
S_5	Yes	Yes	$Omega_{cr2}$	Yes
$S_{6(1)}$	Near	No	*Omega*	Yes
$S_{6(2)}$	No	No	$Omega_{cr1}$	Yes
S_7	Yes	No	$Omega_{cr1}$	Yes

Table 1.2: System models and algorithms.

related topics. Chapter 3 presents the general system model on which our work is based and the redefinitions of Omega for the crash-recovery model ($Omega_{cr1}$ and $Omega_{cr2}$).

In Chapter 4, a first algorithm implementing $Omega_{cr1}$ in the crash-recovery model is given, where processes do not have access to stable storage. Chapter 5 presents three algorithms implementing $Omega_{cr2}$ in three different crash-recovery systems where processes have access to stable storage.

In Chapter 6 we present four new algorithms, which are very efficient from the point of view of communication. The first algorithm, which uses stable storage, implements $Omega_{cr2}$. The second algorithm, which does not use stable storage, implements *Omega*. An adaptation of this second algorithm, which avoids the disagreement of unstable processes by providing instability awareness, is also presented. This algorithm is slightly less efficient, but implements $Omega_{cr1}$ without stable storage. Finally, we present a communication-efficient algorithm that implements $Omega_{cr1}$ without stable storage, using nondecreasing local clocks.

In Chapters 7 and 8 we study the use of Omega in the crash-recovery model. Chapter 7 presents two algorithms implementing two eventually perfect failure detectors, $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$, based on $Omega_{cr2}$. These algorithms use the leader election service provided by the Omega failure detector. In Chapter 8 we propose three aggregator

election and data aggregation algorithms for wireless sensor networks. These algorithms are built on top of our implementations of the $Omega_{cr2}$ failure detector.

Finally, in Chapter 9 we conclude the dissertation and outline our future work.

Chapter 2

Related Work

Contents

2.1 Introduction

In this chapter we review the state of the art in relation to the Omega failure detector. There are hundreds of works that we deem interesting in one way or another but we try not to lose sight of the topic of this thesis: deterministic models and implementations of the Omega failure detector in message passing distributed systems subject to crash-recovery failures. In this regard our work is the first that studies Omega in the crash-recovery model. Nevertheless, Omega has been studied intensively in crash models and hence these studies will form the core of this chapter, together with studies on other unreliable failure detectors.

2.2 Failure Detectors in the Crash Model

Consensus is a canonical problem in which given an initial value at several processes, they agree upon a common value. As we saw in the previous chapter the FLP impossibility result led to the emergence of several lines of research, including the research on unreliable failure detectors.

In the previous chapter we saw the unreliable failure detectors proposed by Chandra and Toueg in [26, 28]. Other works have proposed failure detectors that satisfy different properties to the ones in these papers. In fact, the number of works that implement unreliable failure detectors in a message passing distributed system with crash failures is so great that it is not feasible to review the majority of them here.

This variety of failure detectors is due to the search for weaker and weaker system models in which Consensus or some kind of weaker agreement problem are solvable. In this respect, there are many works that solve Consensus in an asynchronous system with the help of an unreliable failure detector.

The importance of the Omega failure detector derives from the fact that it provides a leader election service that allows circumvention of the FLP result in asynchronous

distributed systems with a minimum amount of synchrony.

As the topic is closely related, we would like to point out several works on solving Consensus with the help of an Omega failure detector or a leader election service [25, 48, 63, 70, 84, 85, 90, 103].

We can find several works in the literature focused on the implementation of Omega in distributed system models subject to crash failures; i.e. where processes fail and do not recover. Most of these are focused on the design of efficient algorithms and the reduction of the level of reliability and synchrony required in the system. A good introductory paper on Omega is [114].

In order to achieve this reduction, two different approaches have been followed. Both approaches consist of enhancements to the system that allow implementing Omega. The first, which is usually called the *timer-based* or *eventually timely link approach* assumes that some communication links in the system are timely and reliable in some way; i.e. they satisfy some time-related properties. The other, called the *message pattern approach*, is a very different approach that does not rely on timing assumptions. The communication requirements of the system are stated in a property of the message exchange pattern, which allows the implementation of Omega.

In the following two subsections we start with algorithms that implement unreliable failure detectors of the classes proposed by Chandra and Toueg [28], which can be transformed into Omega, and then we continue with increasingly evolved implementations of Omega; i.e. implementations that are more efficient and/or require a smaller amount of synchrony.

2.2.1 The Eventually Timely Approach

There are several algorithms implementing Omega using an eventually timely link approach. Among them, we can include most of the algorithms that implement the eventually perfect class of failure detectors, $\diamond\mathcal{P}$, since Omega can be easily obtained from

it. Roughly speaking, $\diamond\mathcal{P}$ satisfies that there is a time after which every correct process maintains the same list of suspected processes. If every correct process chooses as its leader the un-suspected process with the lowest identifier, we have that every correct process trusts the same correct process, the *leader*, thus satisfying Omega. The algorithms implementing $\diamond\mathcal{P}$ in [28, 89, 91] assume that every pair of processes (p, q) is connected by two unidirectional communication links $p \rightarrow q$ and $q \rightarrow p$, with all the links being *eventually timely*, i.e. eventually all messages are delivered within an unknown time bound.

The first $\diamond\mathcal{P}$ algorithm, proposed in [28], periodically sends a quadratic number of messages. In [89] Larrea et al. showed that the four perpetual failure detector classes, $\mathcal{W}$, $\mathcal{S}$, $\mathcal{Q}$, and $\mathcal{P}$, cannot be implemented in the partially synchronous system subject to crash failures considered in their work and in [28], where Consensus is implementable. The conclusion is that it is harder to solve the perpetual failure detector problem than Consensus, at least in the type of partially synchronous systems studied widely. In their paper, Larrea et al. propose a family of algorithms that implements the four classes of failure detectors proposed by Chandra and Toueg that satisfy eventual accuracy in a partially synchronous system; i.e. $\diamond\mathcal{W}$, $\diamond\mathcal{S}$, $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$. The algorithms are based on a strategic arrangement of the processes of the system in a logical ring where, basically, each process only monitors another process by exchanging messages with it. Although the synchrony required is high, they reduce the number of links that periodically carry messages to $2n$ in the worst case, being n the number of processes in the system.

In [91], Larrea et al. presented an optimization of their previous algorithms for the classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$, achieving communication-efficiency, i.e. eventually only n links carry messages periodically. Basically, every process in the algorithm has a predecessor and a successor. A process p sends a message to its successor and expects to periodically receive a *heartbeat* message from its predecessor q. If the process p does not receive the expected message, it informs the predecessor of q, the process r, that it must send

the heartbeat messages to p, instead of to the process q. Later, if p receives a message from q, this means that process q is alive and hence p sends a message to inform r that it must send messages to q again.

To the best of our knowledge, the first algorithm implementing Omega in a partially synchronous system was proposed in [88]. The authors presented an algorithm implementing Omega and through a simple transformation $\diamond\mathcal{S}$, which also requires all links to be eventually timely. Basically, with this algorithm eventually only the correct process p with the lowest identifier sends messages, because the rest of the correct processes will not send messages while receiving periodic messages from p. If the process p fails, after an instability period, the next correct process with the lowest identifier will be elected as the leader, sending periodic messages to the rest of the processes. The number of links that carry messages periodically from the leader process to the rest of the correct processes is $n-1$, which is optimal in this case, i.e. it is communication-efficient.

Aguilera et al. proposed in [5] a communication-efficient Omega algorithm for distributed systems where some unknown correct process must have all its (incoming and outgoing) links eventually timely, while all the other links can be *lossy asynchronous*. Messages sent through a lossy asynchronous link can be lost or arbitrarily delayed. This algorithm is based on rounds. Basically, every process executes rounds (where $r = 1,2,3...$) and in every round only one process is candidate to be the leader. In a round k, the candidate will be the process whose identifier is k *mod* n. Eventually, if the candidate process of a round k communicates in a timely way with the rest of the processes it will remain as leader, and if it does not the round $k+1$ will be initiated with a different process as candidate. In order to synchronize the rounds, the processes exchange a succession of messages, but these messages are not periodic.

In [4], an extended version of [6], Aguilera et al. proposed three algorithms implementing Omega. The first algorithm implements Omega in a partially synchronous

system with at least one *eventually timely source*, which is a correct process whose output links are eventually timely. The rest of the links in the system can be lossy asynchronous. They also show that it is not possible to implement a communication-efficient algorithm in such a system. The second algorithm implements a communication-efficient Omega in a system with at least one eventually timely source and at least one *fair hub*. A fair hub is a correct process such that the links to and from that process are *fair lossy*. These links may lose messages but they satisfy that if we partition messages into types and messages of some type are sent infinitely often, then infinite messages of that type are received.

The third algorithm is communication-efficient and it is implemented in a system with at least one eventually timely source and where the rest of the links are *fair lossy*. We are not going to give the details of the algorithms, but in essence they are based on *punish counters*. A process p with lossy asynchronous and/or fair lossy outgoing links will be punished, i.e. an *ACCUSATION* message will be sent to p when another process detects a missing message from p, incrementing p's punish counter. Processes which have eventually timely outgoing links to the rest of the correct process, e.g. eventually timely sources, will not be punished because their messages will be received timely. With such a mechanism the processes eventually will agree on the correct process with the lowest associated punish counter.

In [7], Aguilera et al. proposed two algorithms for systems in which at most f processes can crash and there is at least one $\diamond f$*-source*, which is a correct process that has at least f eventually timely outgoing links. The first algorithm requires the rest of the links to be fair lossy. Based on the information provided through the fair lossy links and the knowledge of n and f, the algorithm requires only f eventually timely links sending messages periodically. As every process sends messages periodically, the number of links that carry messages forever is $O(n^2)$ in the worst case. The second algorithm requires all the links to be eventually timely and $n > 2f$. Under these conditions, this

algorithm achieves that eventually only one $\diamond f$-*source* sends messages through f eventually timely links. The rest of the processes eventually stop sending messages.

In [95] Malkhi et al. implemented an Omega algorithm that requires a majority of correct processes and the existence of at least one $\diamond f$-*accessible* process, instead of classic eventually timely links. Basically, a correct process p is $\diamond f$-*accessible* if there is a known δ and a time t, such that at any time $t' > t$ there is a set $Q(t')$ of f correct processes, f being the maximum number of processes that can crash, such that any message broadcast by p at time t' will receive a response message from each process in $Q(t')$ by time $t' + \delta$. The key point is the fact that the set of processes $Q(t')$ can change through time. The cost of this algorithm in terms of links used periodically is $O(n^2)$. From this main algorithm, they go on to make an important improvement, presented in a technical report with the same title [96], reducing the number of links that periodically carry messages to $2n$ in the worst case, which is $O(n)$.

The system models proposed by Aguilera et al. in [7] and by Malkhi et al. in [95] are not comparable, and hence we can not establish that one requirement is stronger than the other. The algorithm in [7] requires at least one $\diamond f$-*source* process, i.e. a correct process that has at least f eventually timely outgoing links and no correctness of the f receivers, while the algorithm in [95] requires at least one $\diamond f$-*accessible* process, i.e. a correct process with f possibly changing bidirectional timely links, leading to f possibly changing correct processes. It should be noted that a $\diamond f$-*accessible* process allows for the f processes and f links to change over time, whereas with the $\diamond f$-*source* the set of f links is fixed.

Malkhi et al. justify the relevance of their system model from a practical point of view. The Paxos protocol [84] which is the Consensus protocol, or algorithm, in which its Omega algorithm is based, requires a single leader process to be able to receive f response messages from f different processes in order to *terminate* the Consensus and, similarly, their Omega algorithm requires that a single process, the leader, periodically

receives f response messages from f processes, which could change, to work properly.

Hutle et al. in [78] go a step further in the search for the minimum communication reliability and synchrony required to implement Omega. To this end they combine the concept of $\diamond f$-*source* [7] with the concept of $\diamond f$-*accessibility* [95]. They implement Omega in a system model where all the links between processes are fair lossy, except for the links from at least one process that will be a $\diamond$*moving-f-source*. Basically, a $\diamond$*moving-f-source* is a correct process for which eventually, every time it sends a message to the rest of the processes, at least f messages reach their destination process timely; i.e. in a timely manner. The difference with respect to a $\diamond f$-*source* lies in the set of links that carry timely messages; this is fixed in a $\diamond f$-*source*, while in a $\diamond$*moving-f-source* it can change over time, as occurs with $\diamond f$-*accessibility*.

To the best of our knowledge, up to now the system model proposed in [78] is the weakest of the models proposed that allow solving Omega following the eventually timely approach. For that reason we will define the $\diamond$*moving-f-source* more formally. A process is a $\diamond$*moving-f-source* if it is correct and if there is a known bound δ and a time t, such that at any time $t' > t$ there is a set $Q(t')$ of f correct processes, with f being the maximum number of processes that can crash in a given execution, such that any message sent or broadcast by p at time t' will reach each process in $Q(t')$ by time $t' + \delta$. The key point is the fact that the set of processes $Q(t')$ can change through time.

Among the system models that allow the implementation of Omega based on an eventually timely approach, we can distinguish those in which the initial knowledge of processes is weak. More precisely, the system models presented in [55, 79, 80] assume that the initial knowledge of each process about the system is limited to its identity and the fact that identities are totally ordered. Initially, processes do not know about the identity of the other processes of the system, the number of processes of the system n, or the maximum number of failures during the execution f. In some works, this lack of knowledge of the processes in the system is called *unknown membership*. In the works

reviewed below, the communication between processes is made through an unreliable broadcast primitive. When a process broadcasts a message, it sends the same message through each of its outgoing links to all the processes reachable directly through these links.

In [79], Jiménez et al. proposed an algorithm implementing Omega with *unknown membership* which requires that eventually all correct processes are reachable timely from the correct process with the smallest identifier. The mechanism they use is interesting although the connectivity assumptions are strong. In the worst case with this algorithm $O(n^2)$ links carry messages periodically.

In [80], Jiménez et al. studied in greater depth the unreliable failure detectors in systems with unknown membership. Interestingly, the unknown membership prevents the implementation of a failure detector of any of the eight classes proposed by Chandra and Toueg in [28]. As the authors reveal, it is not possible to implement weak or strong completeness because the processes, (or, more concretely, the modules of the failure detectors at any process) cannot include in their list of suspected processes a process that never sends a message. Such a process will remain unknown until the end of the run and, hence, it will not be suspected by any other process. They also propose an algorithm implementing Omega with unknown membership which requires that eventually all correct processes are reachable timely from some correct process. The rest of the links are considered lossy asynchronous. The algorithm is based on punish counters. Basically, every process periodically sends a list of the processes it considers to be alive, in the beginning only itself, and an associated punish counter that reflects approximately the number of times the known process has been suspected. When a process receives a message it re-sends the message and it includes all the processes that it does not yet know in its membership list, setting a corresponding timer. Then it updates its punish counter, and it punishes itself by incrementing *its* punish counter in its own list if it is not included in the received list. The key to the algorithm is this *self-punishment*.

When a process p receives a punish list and it is not included in it, this means that the sending process cannot receive the messages p sends; hence p must not be the leader, because the leader should reach every correct process in the system. With regard to the cost, as every process re-sends the messages it receives once, in the worst case with this algorithm $O(n^2)$ links carry messages periodically.

Fernández et al. proposed in [55] a system with *unknown membership* where every pair of processes is connected by two unidirectional fair lossy links, except for at least one correct process whose outputs links to the rest of the correct processes must be eventually timely. The algorithm they propose for implementing Omega in such a system is communication-efficient, i.e. the number of links that carry messages forever in the system is $O(n)$.

2.2.2 The Message Pattern Approach

All the approaches previously considered assume partially synchronous system models, where the implementations of Omega use time-outs, and consequently are timer-based. The work in [101] proposes a different approach, called the *message pattern approach*, to implement the failure detector $\diamond\mathcal{S}$, which is easily transformable into Omega. This does not rely on timing assumptions and time-outs but involves a knowledge of the number of the processes in the system (n), the maximum number of processes that can crash (f) and a property *(PR)*.

They define a property *PR* for the message exchange pattern, which makes possible the implementation of Omega in asynchronous distributed systems. We now study the model proposed in greater depth. All the communication links are asynchronous but reliable; i.e they do not alter, create or lose messages. The system's processes broadcast query messages. Once a query has been broadcast the process waits and accepts the first $n-f$ corresponding message responses while discarding the rest of the corresponding messages, if any. A process can only broadcast new queries when the previous ones

have been *terminated*; i.e. when the sender receives $n-f$ response messages. Besides the query-response pattern, Mostéfaoui et al. also defined the following property in the message exchange pattern that we will call *PR*:

There is a correct process p_i and a set Q of f processes such that $\forall p_j \in Q$, there is a time after which p_j crashes or every query of p_j always gets responses from the process p_i, i.e. eventually the response messages of p_i will be among the first $n-f$ responses received by every p_j.

The algorithm works for any value of f, with $1 \leq f < n$. Intuitively, every process p_j periodically sends a query. Once the query is terminated and p_j has received the corresponding $n-f$ responses, where f is the number of processes that did not respond or which had their responses discarded. Then p_j will suspect these f processes in order to achieve completeness. By property *PR*, there will eventually be at least one process p_i for which its responses will be always received among the first $n-f$ messages and will therefore never be suspected. Thus, p_i will eventually never be suspected, the lists of suspected processes will not contain at least one common correct process, satisfying eventual weak accuracy, and hence $\diamond\mathcal{S}$ and indirectly Omega, are implemented. In their work, Mostéfaoui et al. also make a probabilistic analysis in order to show that the behavioural requirements in the message exchange pattern are met with high probability when $f = 1$ with some assumptions. With regard to the cost of the algorithm, in the worst case, the number of links that carry messages forever is $O(n^2)$.

The work of Mostéfaoui et al. in [102] continues the line of research on the Omega failure detector by the definition of behavioural properties for the message exchange patterns. Mostéfaoui et al. implement an explicit Omega algorithm in the model presented in [101], and they rewrite the behavioural property assumed in the system, to the following:

There are a correct process p and a set Q of f processes such that $p \notin Q$ and eventually the response of p to each query issued by any $q \in Q$ is always a winning

response (until -possibly- the crash of q).

As in the work in [101], processes broadcast queries and wait until the receipt of the corresponding responses. The first $n-f$ responses received are called *winning responses* while the rest of the responses, f, are said to be *losing responses*. More precisely, the response messages that reach the receiver after the $n-f$ previous messages, the messages that are lost, and even the responses that are not sent are said to be *losing responses*. The algorithm is based on the previous work, and hence the cost associated in terms of links that carry messages forever is $O(n^2)$.

The work in [104] combines the assumptions proposed in [101, 102], with the assumption in [7] in order to define a weaker system model where Omega is implementable. Subsequently, this hybrid system model combines the time-free assumptions on the behaviour of the message exchange pattern with the synchrony assumptions on the processing speed and message delay. More precisely, the system model requires the following communication assumptions:

There is a correct process p (center of the star) and a set Q of f processes q, $p \notin Q$, such that, eventually, either 1) each time it broadcasts a query, q receives a response from p among the $n-f$ first responses to that query, or 2) the channel from p to q is timely. (The processes in the set Q can crash).

Similarly to the work in [102], the links in the system are reliable although a priori asynchronous, the value of f is such that $1 \leq f < n$, and the number of links that periodically carry messages is $O(n^2)$. The paper also includes an improved algorithm where the communication properties are satisfied by directed *paths* connecting *p* and *q* instead of direct links. Paths will be explained in more detail in Section 3.1, but basically a path is composed of connected correct processes.

To the best of our knowledge, up to now the work in [56] presents the weakest system that allows the implementation of Omega following the message pattern approach. The authors present two algorithms that implement Omega in an asynchronous system that

satisfy the properties (assumptions) $A+$, in the first algorithm, and A in the second. In the system the links do not have any timing assumptions and are reliable; however, it is said in this work that the algorithms are also correct considering fair lossy links.

The model with the A property, the weakest model proposed so far, is characterized by the notion of *intermittent rotating f-start*. An *f-star* is a set of $f+1$ processes where a process p is the center of a star and the rest of processes are the points of the star. The property that satisfies the *f-star*, denoted by A in the paper, is described in two steps. In the first step, the property $A+$ is defined; this is stronger than A and satisfies the following:

There is a correct process p and a round number RN_0 such that, for each $rn \geq RN_0$, there is a set $Q(rn)$ of f processes such that $p \notin Q(rn)$ and for each process $q \in Q(rn)$ either (1) q has crashed, or (2) the message $ALIVE(rn)$ sent by p is received by q at most δ time units after it has been sent (the corresponding bound δ can be unknown a priori), or (3) the message $ALIVE(rn)$ sent by p is received by q among the first $n-f$ $ALIVE(rn)$ messages received by q (i.e. it is a winning message among $ALIVE(rn)$ messages received by q).

It is noteworthy that the set Q can change from one round rn to another. As we can see, the definition is a combination of assumptions presented in previous works, in both timer-based and message pattern approaches. If we take into account (1) and (2), the property satisfied is a $\diamond$*moving-f-source* [78]. However, taking into account (1) and (3) the property satisfied is a moving version of the message exchange pattern assumption presented in [104]. The *f-star* is formed by the f processes in $Q(rn)$, and the center p. The set $Q(rn)$ can change at each round number, while p can not, so we say that p is the center of a $\diamond$*rotating-f-star*.

As has previously commented, they proved that in order to implement Omega the property A, which is weaker than $A+$, is sufficient. The property A assumes that p only will be the center of the $\diamond$*rotating-f-star* an infinite subset of rounds that is not known

a priori. They called this configuration a $\diamond$*intermittent-rotating-f-star*.

2.3 Failure Detectors in the Crash-Recovery Model

The related works commented on so far consider distributed systems subject to crash failures, also called crash-stop failures; i.e. system models in which once a process crashes it does not recover again. As we mentioned in the previous chapter, there are more types of failures such as crash-recovery, omission and Byzantine. One of the most interesting is the crash-recovery model, where processes can crash and later recover by rejoining the computation from a recovery point. This behaviour is especially common for long-lived applications such as distributed operating systems, grid computing, or web services and has been formalized as a failure model called crash-recovery. Hence, in the crash-recovery model, processes can crash multiple times. After crashing (and before crashing the next time), a process recovers from a predefined state.

The crash-recovery failure model is a strict generalization of the crash failure model; i.e. every faulty behaviour allowed in the latter is also possible in the former. This means that any impossibility result for the crash model also holds in the crash-recovery model. As a result, an algorithm designed for the crash-recovery model will work correctly in a similar system model where processes are subject to crash failures, as this kind of behaviour is also permitted in a system subject to crash-recovery failures. However, this is not true in the other direction, due to the additional faulty behaviour considered in the crash-recovery model.

When implementing unreliable failure detectors in a distributed system subject to crash-recovery failures we must deal with new difficulties. The first is related to the behaviour of the processes. In the crash-recovery model the processes are classified into *correct* processes and *incorrect* processes. Correct processes, which some authors call *good* processes [3], are the processes that (eventually) do not crash:

- Always up. The subset of processes that never crash.
- Eventually up. The subset of processes that, after crashing and recovering a finite number of times, remain up forever.

Incorrect processes, which some authors call *bad processes*, are the ones that either crash and recover infinitely often or do not recover after a crash:

- Eventually down. The subset of processes that, after crashing and recovering a finite number of times, remain down forever. Processes that never start their execution are included in this subset.
- Unstable. The subset of processes that crash and recover an infinite number of times.

As we can foresee unstable processes require change in perspective when designing a new unreliable failure detector, due to the infinitely repeated crashes. In this regard, we have two options in the crash-recovery model. We could include the process in the list of suspected processes every time we detect its failure and remove it from the list every time the process recovers. However, this option has an important drawback. It could happen that repeatedly:

(1) an application considers an unstable process as operational (correct);

(2) the application delegates a procedure or function to the unstable process; and

(3) the unstable process crashes before finalizing the task.

This behaviour will slow down the application and can even block the application if it does not deal with the problem properly.

On the other hand, we would like the failure detector to suspect unstable processes permanently, because we know they are incorrect. However, we have to take into account the existence of the eventually up processes, which after crashing and recovering

a finite number of times will remain up forever, with a correct behaviour. Then, the question is how can we distinguish an unstable process from an eventually up one? The answer is simply that we cannot. This is because we cannot predict the future behaviour of a process. Thus, at any time we do not know if a process is eventually up and has definitely recovered or it is an unstable process that will crash and recover again and again.

Another interesting issue is that every time a process crashes it loses its entire local volatile state, i.e. the values of its variables, unless they are recorded in stable storage; which is considered slow and expensive, especially if every process saves its local state to stable storage periodically. In this regard, Aguilera et al. in [3] proved that it is possible to implement failure detectors in the crash-recovery model *without*, and obviously with, using stable storage.

Basically, by the use of stable storage we can weaken the synchrony, failure and communication requirements of the system in which the failure detector is implemented at the cost of *some* reading and writing operations in stable storage. However, the reading and writing operations are expensive and they are not always available in real systems.

With regard to communication, we also have a new issue. How reliable is the link to a process that can crash and recover? A common assumption is the one made in this thesis: a message sent to a process that has crashed is lost. If a process p sends a message to a process q that is up, but q crashes before completing its reception, the message is also lost. Reliable links, such as the eventually timely links explained in the next chapter, guarantee that a sent message is only received if the receiver does not crash.

Another option is the use of *stubborn communication channels* [69]. These channels, or links, allow reliable communication by resending a message until the message is received or the sender crashes. In order to avoid buffer overflow the definition assumes

that it is not possible to send a message m' through the stubborn channel if the reception of a previous message m is not assured. This work [69] presents an interesting review about communication links [17, 94] and their properties.

Surprisingly, the number of works that consider a crash-recovery failure model is small compared to other less realistic failure models such as the omission or the crash failure models. We can find several works that provide Consensus algorithms for the crash-recovery model by relying on the existence of an unreliable failure detector. A good introductory work on Consensus in the crash and crash-recovery failure model is [66].

In the works [46, 76, 107], the authors define adaptations of the $\diamond\mathcal{W}$ and/or $\diamond\mathcal{S}$ unreliable failure detector classes for the crash-recovery model and implement Consensus protocols based on these new classes. However, no algorithm implementing the failure detectors is provided.

Dolev et al. in [46] consider crash-recovery as a special case of omission failures. The Consensus algorithm they provide is not designed to handle unstable processes that may intermittently communicate with correct processes. In order to prevent the data loss inherent to the process crashes, they consider that at every state transition the whole state of the algorithm is saved to stable storage.

In [107], Oliveira et al. propose a Consensus algorithm that requires a failure detector that eventually suspects unstable processes forever. It has been shown in [3] that such a failure detector implies a loss of performance in many scenarios. As in [46] there is an intensive use of stable storage due to the storage of the whole state at every transition.

Hurfin et al. in [76] also define a failure detector that eventually suspects unstable processes forever. The writing to stable storage is done once at most for each round and only a small part of the state is saved.

The drawback caused by the unreliable failure detectors in [76, 107] can be over-

come by modifying them. In his thesis [106] Oliveira defined a new unreliable failure detector class, denoted by $\diamond\mathcal{S}_r$, which satisfies eventual weak accuracy and *recurrent* strong completeness. It has been proved that the aforementioned algorithms are correct with this failure detector. The property is defined as follows:

- *Recurrent strong completeness*. Every incorrect process is infinitely often suspected by every correct process.

Freiling et al. in [61] focus on the reuse of existing algorithms from the crash failure model. The Consensus algorithms proposed use unreliable failure detectors of the classes $\mathcal{P}$ and $\diamond\mathcal{P}$, adapted to crash-recovery model. The adapted $\mathcal{P}$ failure detector class satisfies the following properties:

- Strong completeness: Every incorrect process is suspected infinitely often by every correct process.
- Eventually up completeness: Eventually, every correct process is not suspected any longer by every correct process.
- Strong accuracy: No process is suspected before it crashes.

The adapted $\diamond\mathcal{P}$ unreliable failure detector class satisfies:

- Strong completeness.
- Eventually up completeness.
- Eventually strong accuracy: Correct processes are only finitely often suspected.

The approach they follow consists of partly emulating a crash system on top of the crash-recovery system in order to execute a Consensus algorithm designed for the crash failure model. In the paper, they first provide a set of minimality and impossibility results for the solvability of Consensus in the crash-recovery model and then they propose

three modular algorithms that allow Consensus to be solved in the crash-recovery model for the selected failure detectors with weak system assumptions. The first algorithm allows Consensus to be solved in a system with unavailability of stable storage, at least one always up process and the help of a $\mathcal{P}$ failure detector. The second algorithm, also without stable storage, requires a majority of always up processes and a $\diamond\mathcal{P}$ failure detector. Finally, the third algorithm requires a $\diamond\mathcal{P}$ failure detector, a majority of always up or eventually up processes, and the recording of some information used by the crash Consensus algorithm to stable storage.

In their seminal work [3], Aguilera et al. studied the problem of Consensus and failure detection in the crash-recovery model. With regard to the use of stable storage, they stated that it is possible to solve Consensus in the crash-recovery model if $n_a > n_b$ with the help of an adapted $\diamond\mathcal{S}$ failure detector, $\diamond\mathcal{S}_e$, where n_a is the number of always up processes and n_b is the number of incorrect processes, even if $n_a < \frac{n}{2}$. An algorithm that solves Consensus under the aforementioned conditions is given. Curiously, this necessary condition remains if only the proposal and decision values can be saved to stable storage. For systems where it is possible to save more information to stable storage, they provide an implementation of a Consensus algorithm that works with a majority of correct processes, relying on a failure detector of the class $\diamond\mathcal{S}_u$, adapted from $\diamond\mathcal{S}$. In the algorithm, every process accesses stable storage twice each round. The information stored is a round number, an estimate and its corresponding timestamp.

As far as we know, the work in [3] is the only one that deals with the implementation of deterministic unreliable failure detectors in message passing asynchronous or partially synchronous system models subject to crash-recovery failures. The system model assumed is an extension of the *M3* system model [28] presented in the previous chapter. In this extended *M3* model, processes can crash and recover, and messages sent to incorrect processes can be lost. The links considered are eventually timely. In this work Aguilera et al. first showed that strong completeness, which is satisfied by $\diamond\mathcal{S}$,

involves a loss of efficiency when running an algorithm that solves Consensus as those in [28, 46, 76, 107]. Subsequently, they proposed two new classes of unreliable failure detectors that deal with unstable processes in a different way. Innovatively, the output of a failure detector module at every process p is made up of two lists instead of one. The first list, as usually, is the list of trusted processes at the current time t. The second list provides an estimate of the number of times that each process in the list of trusted processes has crashed and recovered so far. This estimate is called the epoch number.

They provide an algorithm implementing a failure detector of the class $\diamond\mathcal{S}_e$ that, without going into too much detail, satisfies the following properties:

- Monotonicity: At every correct process, eventually the epoch numbers are non-decreasing.

- Completeness: For every incorrect process p and for every correct process q, either eventually q permanently suspects p or p's epoch number at q is unbounded.

- Accuracy: For some correct process p and for every correct process q, eventually q permanently trusts p and p's epoch number at q stops changing.

With regard to the implementation of $\diamond\mathcal{S}_e$, basically, every process maintains a list of trusted processes and also maintains a list of the epoch numbers of the rest of the processes by counting the *RECOVERED* messages it receives. During initialization and upon recovery, a process p sends a *RECOVERED* message to the rest of the processes. When a process receives the *RECOVERED* message from p, it increments the local epoch number associated with p. Similarly to the algorithm in [28], every process p periodically sends an *ALIVE* message. Then p checks the reception of messages. If p does not receive an *ALIVE* message from a process q by the time it expects to receive it, p removes q from the list of trusted processes. If p receives a message from a process that is not trusted, p increments its associated counter and includes it in the list.

In terms of periodically sent messages, in the worst case there will be n processes sending $n-1$ messages, making a total of $O(n^2)$ messages sent periodically. From the point of view of links, in the worst case $O(n^2)$ links carry messages forever.

The properties satisfied by the $\diamond\mathcal{S}_e$ failure detector class only refer to correct processes, and hence the output of unstable processes is not restricted in any way. However, it would be desirable for the erroneous suspicions of unstable processes to be limited by satisfying some degree of accuracy. For this reason, they define a new failure detector class $\diamond\mathcal{S}_u$ that satisfies the following properties:

- Monotonicity.
- Completeness.
- Strong Accuracy: For some correct process p: (a) for every correct process q, eventually q permanently trusts p and p's epoch number at q stops changing; and (b) for every unstable process u, eventually whenever u is up u trusts p and p's epoch number at u stops changing.

As in the previous class, the output at a failure detector module consists of a list of trusted processes and a list of their respective epoch numbers. With respect to implementation, a transformation from $\diamond\mathcal{S}_e$ to $\diamond\mathcal{S}_u$ is provided. The transformation does not use stable storage and requires a majority of correct processes.

The algorithm works roughly as follows: During initialization every process includes all the processes in its trusted list. Every process p has access to a failure detector of the class $\diamond\mathcal{S}_e$ and periodically sends the two lists it provides to the rest of the processes. Therefore, every process will periodically receive the trusted list and epoch number list from at least $\lceil\frac{n}{2}\rceil$ processes because, by definition, there is a majority of correct processes. When process p realizes that a process q is not included in a majority of the trusted lists provided by the $\diamond\mathcal{S}_e$ failure detectors, p removes q from its trusted list.

With regard to the epoch numbers, every process p increases the epoch number associated with a process q when p detects, through the lists provided by the $\diamond\mathcal{S}_e$ failure detectors, that q is suspected or it is trusted but its epoch number has been increased by a majority of processes.

In terms of periodically sent messages, we must add the messages required for the transformation to the messages required by the algorithm that implements $\diamond\mathcal{S}_e$. Hence, although the cost in periodically sent messages is higher, it is still $O(n^2)$ in the worst case. From the point of view of links, in the worst case $O(n^2)$ links carry messages forever.

2.4 More about Failure Detectors

In this section we will review some works about failure detectors that, although interesting, are less directly related to the topic of this thesis.

The use of failure detectors to allow the implementation of quiescent algorithms for reliable communication in a distributed system with crash failures and lossy links is studied in [2]. Briefly, an algorithm is quiescent if eventually it stops sending messages. As a result, they propose a new failure detector (*heartbeat*, denoted by *HB*) that does not rely on time-outs. The output at a process p of a *HB* module consists of a vector of counters, one for each neighbour q of p. If q crashes this counter is bounded, otherwise it is not bounded. The implementation of the failure detector is intuitive. Processes send periodic messages and when a process p receives a message from a process q, p increments the counter associated with q in its output vector.

Delporte-Gallet et al. in [42] propose a set of distributed algorithms that implement a leader election service in systems where processes are subject to crash failures. The communication assumptions for each distributed system varies from a system with all its links eventually timely to a system with at least one *eventually timely source* [4].

The paper focuses on *self-stabilizing* [43] and *pseudo-stabilizing* [24] algorithms, and provides communication-efficient implementations when possible.

Stabilization is a general technique that allows algorithms to tolerate *transient* failures. Informally, an algorithm that is self-stabilizing will *remain* in a correct state at a finite time independently of its initial state. An algorithm that is pseudo-stabilizing will *end up* in a correct state at a finite time independently of its initial state. It should be noted that in many cases stabilizing algorithms support dynamic topological changes.

The *Quality of Service*, denoted by QoS, has been addressed in some works. Chen et al. in [34] specify the QoS of a crash failure detector in terms of three primary metrics and four derived metrics. We explicate the primary metrics very briefly as follows:

- *Detection time*. The time that elapses from the crash of a process p to the time when another process q starts suspecting p permanently.
- *Mistake recurrence time*. This measures the time between two consecutive mistakes.
- *Mistake duration*. This measures the time it takes the failure detector to correct a mistake.

They also provide a new implementation of a failure detector and justify its optimality in terms of the proposed metrics.

2.5 Solving Consensus

The Consensus problem has been studied intensively in a variety of system models with different techniques. In this section we will mention some relevant works that have not been reviewed previously.

In [87] we find the *Byzantine Generals Problem* defined as follows:

A commanding general must send an order to his $n-1$ lieutenant generals such that:

- All loyal lieutenants obey the same order.
- If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

This definition expresses the Consensus problem in a system model where incorrect processes behave maliciously, i.e. processes are subject to Byzantine failures. Although it is not very orthodox, due to its originality, we will let the authors summarize their paper by citing their abstract:

"Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one of more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed."

More interesting results about the Byzantine generals problem can be found in [44].

Schiper in [117] defined the notion of *latency degree*, the minimal number of communication steps needed to solve Consensus, and provided a Consensus algorithm with a latency degree of 2, relying on a failure detector of the class $\diamond\mathcal{S}$. Hurfin and Raynal in [77] presented another Consensus protocol based on $\diamond\mathcal{S}$ which is very efficient when the underlying failure detector makes no mistake (a common case in practice).

Chapter 3

General System Model

Contents

3.1 Definition of the General System Model

In this section we present the general crash-recovery system model S, where we study the Omega failure detector. Starting from this system model we either add or slightly modify some system assumptions, leading us to different specific systems in which we propose distributed algorithms implementing the Omega failure detector. More precisely, in the following chapters we will define the specific systems S_1 to S_8, relating them to the general system model S of this chapter.

We also redefine Omega for the crash-recovery model. Depending on whether or not stable storage is used, the level of agreement of unstable processes with respect to correct processes varies. In a system where processes have access to stable storage we can implement a stronger property, that we denote by $Omega_{cr2}$, otherwise we only can implement a weaker property, that we denote by $Omega_{cr1}$.

System composition

We consider a partially synchronous distributed system S composed of a finite and totally ordered set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of $n > 1$ processes that communicate only by sending and receiving messages. The process identifiers do not need to be consecutive. Usually, we will use p, q, r, ... to denote processes. By default, every process knows *a priori* the identity of the rest of the processes. We will call this feature *known membership*. However, in some systems this will not be the case and hence we will have *unknown membership*. In these cases it will be adequately pointed out.

Clocks

We assume the existence of a virtual discrete global clock, although processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers. In our partially synchronous system S, in every execution there are bounds

on relative process speeds and on message transmission times; however, these bounds are not known and they hold only after some unknown time, the system-wide Global Stabilization Time *GST*, where $GST \in \mathcal{T}$.

Every process has a discrete local clock that can accurately measure intervals of time. These clocks are not necessarily synchronized and the range of these clocks' ticks is also the set of natural numbers.

Timers

Processes have access to local timers that can be implemented easily with the local clock. A process can set a local timer to a natural number, called the *time-out*, and start it. This will usually be done in the algorithms with the instruction: *reset* $Timer_x$ *to* $Timeout_y$. Once a timer has been started, it decreases by one for every time unit until it reaches 0. At this moment it is said that the timer has expired.

Crashes and recoveries

Processes can crash and may later recover. In order to formalize failures we define a *failure pattern* F as a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that are not functioning at time t. A process p *is up* at time t if $p \notin F(t)$. A process p *is down* at time t if $p \in F(t)$. We say that a process p *crashes* at time t if p is up at time $t-1$; i.e. $p \notin F(t-1)$ and p is down at time t; i.e. $p \in F(t)$. On the other hand, if p is down at time $t-1$ and up at time t, we say that p *recovers* at time t.

When a process crashes, it stops functioning and loses the contents of all its variables that are not stored in stable storage, the timers are stopped and their previous values are lost. When a process recovers, it starts the execution of the algorithm.

Stable storage

In the definition of S we do not specify whether or not processes are able to write to stable (persistent) storage. Some of the systems based on S, e.g. the system in Chapter 4, do not consider stable storage while other systems, such as the ones proposed in Chapter 5, consider that processes can write to stable storage. In every specific system we indicate whether processes have access to stable storage or not. The use of stable storage is discussed in greater depth in Chapter 5.

Types of processes

In every run of S we can distinguish three disjoint subsets of processes according to the failure pattern F:

(1) *Eventually up*. This is the subset of processes that, after crashing and recovering a finite number of times, remain up forever; i.e. they do not crash any more. Processes that never crash are included in this subset. Formally:
$\exists t \in \mathcal{T} : \forall t' > t : p \notin F(t')$.

(2) *Eventually down*. This is the subset of processes that, after crashing and recovering a finite number of times, remain down forever; i.e. they do not recover any more. Processes that never start their execution are included in this subset. Formally:
$\exists t \in \mathcal{T} : \forall t' > t : p \in F(t')$.

(3) *Unstable*. This is the subset of processes that crash and recover an infinite number of times; i.e. there is not a time after which either they remain up forever or they remain down forever. Formally:
$(\nexists t_1 \in \mathcal{T} : \forall t' > t_1 : p \notin F(t')) \wedge (\nexists t_2 \in \mathcal{T} : \forall t'' > t_2 : p \in F(t''))$.

By definition, processes in (1) are *correct*, and processes in (2) and (3) are *incorrect*. We assume that the number of correct processes in the system in any run is at least one.

Processing speed

Processes execute by taking atomic steps. We assume the existence of a lower bound σ on the number of steps per unit of time taken by any process. This bound does not need to hold from the beginning but from GST, the Global Stabilization Time. Moreover, the bound σ may vary for every run. For simplicity, we will assume that each instruction of the algorithms represents one step. We will also assume that the local processing time, related to σ, is negligible with respect to message communication delays.

Finally, we will assume that all the system's processes start the algorithm at approximately the same time. Although actually this is not necessary, the explanations and correctness proofs of the algorithms will be more intuitive.

Communication links

Processes communicate with each other by sending messages through direct links. The network is fully connected: for every pair of processes $p \neq q$ there is a direct link from p to q and another direct link from q to p. A link from a process p to any other process is an *output* link of p and a link from any process to p is an *input* link of p.

We assume that messages are unique in the sense that processes can determine whether a received message is a duplicate of a previously received message. This can be achieved, for example, by including the sender process identifier and a timestamp, provided by its local clock, in each message. With regard to the timeliness and reliability properties, we consider the following three types of links [4, 6]:

(a) *Eventually timely link,* where there is an unknown bound δ on message delays and an unknown Global Stabilization Time, $GST \in \mathcal{T}$, such that if a message m

is sent through the link at a time $t \geq GST$, then the message m is received by time $t + \delta$ if the receiver process is up.

(b) *Lossy link*, where the link can lose or delay an arbitrary number of messages.

(c) *(Typed) Fair lossy link*, where, assuming that each message has a type, if for every type infinitely many messages are sent then infinitely many messages of each type are received if the receiver process is correct.

Note that the time GST and the bound δ can vary for each run. Furthermore, if we knew these values *a priori*, we could easily implement a failure detector that satisfies perpetual accuracy, i.e. that does not make erroneous suspicions, in the crash model assuming that all links are eventually timely. Basically, we can construct a failure detector where every process p in the system waits until the time GST arrives and then sets timers with respect to the rest of the processes $q \neq p$ to δ. If the timer for a process q expires it means that q has crashed because otherwise any message sent must have arrived.

We consider that no link in S modifies its messages or generates spontaneous messages. However, it may deliver them out of order. More precisely, links of any type in S satisfy the following integrity property:

Property 1 (*Integrity***)** *A message m is delivered to q from p only if p sent m to q.*

Communication primitives

To send messages, processes have atomic sending primitives that allow them to send the same message m through the required outgoing link. For example, if p executes *send m to q*, then m will be sent through the outgoing link from p to q in one step. Similarly, if p executes *send m to all* or *broadcast m*, p will send the message m through all its outgoing links in one step.

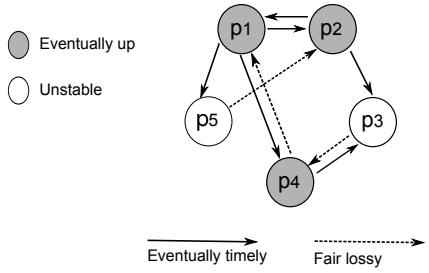

Figure 3.1: Example of paths.

Connectivity requirements

The general system model S does not specify the particular connectivity considered in the systems proposed in this thesis. Each specific system in which we implement Omega has its own connectivity requirements. A common communication requirement in this dissertation is the existence of a *path of links* of a specific type, e.g. eventually timely, from one or more processes to other processes. For example, the system S_1 presented in Chapter 4 has the following connectivity requirement:

There is a correct process p such that there is an eventually timely path from p to every correct and every unstable process.

The above mentioned eventually timely paths are formed by correct processes connected through eventually timely links. As eventually timely paths depend on eventually timely links, the communication assumptions that rely on eventually timely paths are achieved after an unknown but finite time.

Now, we define the concept of *path of links* more formally. A path from p to q, $p \Longrightarrow q$, is a directed graph composed of processes as vertexes and links as connectors, denoted by $\rightarrow$, of the form $p \Longrightarrow q \equiv p \rightarrow q_1,\ q_1 \rightarrow q_2,\ ...,\ q_n \rightarrow q$, such that the intermediary processes $\{q_1, q_2, ..., q_n\}$ are correct and all the links are of the required type or stronger; e.g. eventually timely. A direct link from p to q is also considered a path: $p \rightarrow q \equiv p \Longrightarrow q$.

Observe that processes p and/or q could be unstable. In this case, the connectiv-

ity requirements that affect unstable processes, e.g. a path that starts or finishes at an unstable process, only hold when the unstable process is up.

In the example in Figure 3.1 processes p_1, p_2 and p_4 are correct, i.e. eventually up, and processes p_3 and p_5 are unstable. There is an eventually timely path from the processes p_1 and p_2 to all correct and unstable processes. Besides, there is a fair lossy path from every unstable process to every correct process.

3.2 Omega in the Crash-Recovery Model

During their study of the Consensus problem and the use of unreliable failure detectors to solve it, Chandra et al. defined in [26] a failure detector called Omega for the crash model. Informally, the output of the Omega failure detector module at a process p is a single process q that p currently considers to be correct (we say that *p trusts q*). Eventually, this output must be the same at every correct process and must correspond to a correct process. More formally, Omega satisfies the following property:

Property 2 (*Omega***)** *There is a time after which every correct process always trusts the same correct process.*

Note that the output of the failure detector module of Omega at a process p may change over 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.

Typically in our algorithms the trusted process at a process p, which is the output of the Omega module, is held in the local variable $leader_p$. In order to export this variable, every algorithm includes a function *leader()*, which for simplicity has been omitted from the pseudocode of the algorithms. This function returns the identity of the process trusted by p's Omega module at a given time: the value of the variable $leader_p$, if and only if the variable has been set to any value by the algorithm. When an application

calls the function *leader()*, if $leader_p$ has not been set (i.e. p does not trust any process), the special value $\perp$ is returned.

The definition of Omega was proposed for the crash model, and hence it does not consider unstable processes. In the crash-recovery model, it is not possible for a process to determine whether it is: a correct process; an eventually down, but still up, process; or even an unstable, but up, process. Note that if we keep Omega as is for the crash-recovery model then unstable processes are allowed to disagree with correct processes, which can be a serious drawback. For instance, when solving Consensus, termination of Consensus cannot usually be guaranteed if correct processes select a leader that is different from the one selected by unstable processes. Hence, it would be desirable in a crash-recovery system that all processes which are up eventually agree on a common correct leader process. In order to express this we first redefine the property that Omega must satisfy, adapted to the crash-recovery model without stable storage:

Property 3 ($Omega_{cr1}$**)** *There is a time after which (1) every correct process always trusts the same correct process l, and (2) every unstable process, when up, always trusts either $\perp$ or l. More precisely, upon recovery an unstable process will first trust $\perp$ (i.e. it does not trust any process), and –if it remains up for sufficiently long– it will then trust l until it crashes.*

If we consider the use of stable storage by processes, then new possibilities arise. In this regard, the quality of the agreement of unstable processes with the correct ones will depend on whether or not of stable storage is used. Intuitively, the use of stable storage allows unstable processes to eventually agree from the beginning of their execution by reading the identity of the leader from stable storage, while the absence of stable storage forces unstable processes to communicate with some correct process(es) in order to learn the identity of the leader.

We redefine Omega for crash-recovery models where processes have access to stable storage as follows:

Property 4 ($Omega_{cr2}$**)** *There is a time after which every process that is up, either correct or unstable, always trusts the same correct process.*

Note that $Omega_{cr2}$ is stronger than $Omega_{cr1}$.

Chapter 4

Omega in Crash-Recovery without Stable Storage

Contents

4.1 Introduction

In this chapter we propose an algorithm implementing $Omega_{cr1}$ for the crash-recovery model in a distributed system where processes do not have access to stable storage. We consider this aspect interesting for a number of reasons. First of all, the use of stable storage is expensive in time. When designing an algorithm, we must take into account that the time required to execute an instruction that accesses stable storage, i.e. an external device, can be several orders of magnitude higher than for the execution of an instruction that does not access it; in this regard, the algorithms presented in Chapter 5 use stable storage, but each process only accesses stable storage a small number of times upon recovery. Secondly, we cannot assume that all the real devices that may execute an Omega algorithm will have access to stable storage. Finally, from a more theoretical point of view, in our search of weaker distributed system models for implementing efficient Omega algorithms, we must consider systems in which processes do not have access to stable storage because this assumption is weaker than the opposite. As we will see, the algorithm presented in this chapter assumes a majority of correct processes in order to implement $Omega_{cr1}$ (Property 3).

In Chapters 4, 5 and 6 we address the implementation of the Omega failure detector in crash-recovery systems. The outline followed will be similar for each algorithm. First, we present the specific system assumptions, then the pseudocode of the algorithm executed at every process p with the required explanations, and finally a proof or proof sketch of the correctness of the algorithm.

Some of the algorithms share some names of constants and variables. To avoid redundancy, when explaining an algorithm, the explanations of these ones can be omitted if they have been defined previously. If a constant or variable has been defined in two or more places with different meanings the corresponding definition is the latest one, unless otherwise stated.

The rest of this chapter is organized as follows: firstly, in Section 4.2 we present the

system S_1 and an algorithm implementing $Omega_{cr1}$ in it. Secondly, in Section 4.3, we discuss the eventual timeliness of fair lossy links.

4.2 An Algorithm for System S_1

4.2.1 Specific System Assumptions in S_1

The system S_1 corresponds to the general system model S, defined in Chapter 3, with some additional assumptions.

With regard to connectivity, in S_1 we make the following assumptions:

(1) There is a correct process p such that there is an eventually timely path from p to every correct and every unstable process.

(2) For every correct process $q \neq p$, there is a fair lossy path from q to p.

(3) For every unstable process u, there is a fair lossy link from u to some correct process.

We also assume that a majority of processes are correct.

4.2.2 The Algorithm

In this section we propose a distributed algorithm implementing $Omega_{cr1}$ in system S_1. Figure 4.1 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

With respect to the variables, every process p has a $leader_p$ variable containing its trusted process, initialized to $\perp$, and a $Candidates_p$ set containing the processes from which p will choose $leader_p$, initialized to Π. In addition, p has a time-out with respect to every other process, the array $Timeout_p$ which is initialized to η, and a counter of the

approximate number of times that each process has recovered or has been suspected, the array $Punish_p$.

The time constant η is used in all the algorithms and indicates the rate at which periodic messages will be sent by processes. Its specific value is very important if we want to run the algorithms in a real environment according to real system requirements. If η is too small we will have a flood of messages, and hence the messages would possibly be queued or lost and the timers would expire. This should be avoided at all costs. On the other hand, a higher value of η implies a slower convergence of the algorithm.

Below, we elaborate on the functioning of the algorithm. During the execution of the *Initialization*, and upon recovery, the process p sends a $RECOVERED$ message to all the processes, sets $timers\_active$ to $FALSE$, and starts the four tasks of the algorithm. Note that all the timers of p are inactive. If p does not crash, the reception of enough $ALIVE$ messages is guaranteed by the assumption that a majority of processes are correct in S_1. In Task 1, p periodically sends an $ALIVE$ message containing $Punish_p$ to all processes. In Task 2, when p receives a $RECOVERED$ message from q, p increments $Punish_p[q]$.

In Task 3, when p receives an $ALIVE$ message from $q \neq p$ that was not received previously, p resends the message to all the processes and updates $Punish_p$ with $Punish_q$ by taking the highest value for each component of the vector. Then, p also updates all its time-outs, taking the highest value of the current time-outs and $Punish_p[p]$.

Finally, if p has so far received $ALIVE$ messages from a majority of processes $\lceil \frac{n+1}{2} \rceil$, if the timers are not yet active, p resets all its timers for the first time after the recovery, and sets $timers\_active$ to $TRUE$. In addition, p includes q in $Candidates_p$, if required, and increments $Timeout_p[q]$. Then it resets $Timer_p(q)$ and calls the procedure $update\_leader()$ in order to update $leader_p$ to the process in $Candidates_p$ with the minimum associated punish counter in $Punish_p$. In Task 4, when $Timer_p(q)$ expires, p increments the associated punish counter, $Punish_p[q]$, removes q from $Candidates_p$ and

Every process p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow l$ such that $Punish_p[l] = min\{Punish_p[q]\}$, $\forall q \in Candidates_p$, using identifiers to break ties

Initialization:
(2) $leader_p \leftarrow \perp$
(3) $Candidates_p \leftarrow \Pi$
(4) $\forall q \neq p : Timeout_p[q] \leftarrow \eta$
(5) $\forall q : Punish_p[q] \leftarrow 0$
(6) send ($RECOVERED$, p) to all processes
(7) $timers\_active \leftarrow FALSE$
(8) **start tasks** 1, 2, 3 and 4

Task 1:
(9) **loop forever**
(10) send ($ALIVE$, p, $Punish_p$) to all processes
(11) wait(η)

Task 2:
(12) **upon reception of** message ($RECOVERED$, q) **do**
(13) $Punish_p[q] \leftarrow Punish_p[q] + 1$

Task 3:
(14) **upon reception of** message ($ALIVE$, q, $Punish_q$) with $q \neq p$ for the first time **do**
(15) send ($ALIVE$, q, $Punish_q$) to all processes
(16) $\forall r : Punish_p[r] \leftarrow \max\{Punish_p[r], Punish_q[r]\}$
(17) $\forall r : Timeout_p[r] \leftarrow \max\{Timeout_p[r], Punish_p[p]\}$
(18) **if** p has received so far $ALIVE$ from a majority of processes **then**
(19) **if** $timers\_active = FALSE$ **then**
(20) $\forall q \neq p$: reset $Timer_p(q)$ to $Timeout_p[q]$
(21) $timers\_active \leftarrow TRUE$
(22) **end if**
(23) **if** $q \notin Candidates_p$ **then**
(24) $Candidates_p \leftarrow Candidates_p \cup \{q\}$
(25) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(26) **end if**
(27) reset $Timer_p(q)$ to $Timeout_p[q]$
(28) $update\_leader()$
(29) **end if**

Task 4:
(30) **upon expiration of** $Timer_p(q)$ **do**
(31) $Punish_p[q] \leftarrow Punish_p[q] + 1$
(32) $Candidates_p \leftarrow Candidates_p - \{q\}$
(33) $update\_leader()$

Figure 4.1: An algorithm implementing $Omega_{cr1}$ in system S_1.

calls $update\_leader()$.

With this algorithm, eventually all the processes that are up will have in $leader_p$ either $\perp$, which indicates that they have not yet received $ALIVE$ from a majority of processes, or the common correct leader l. An important detail is that at a given process p no timer is activated and, hence, no timer expires until p has received an $ALIVE$ message from a majority of processes, preventing erroneous suspicions from unstable processes. After the reception of $ALIVE$ messages from a majority of processes at an unstable process u, it is guaranteed that at least one message has been sent from a correct process, so u will have (1) $Punish_u$ such that l is chosen as leader, and (2) $Timeout_u[l]$ such that $Timer_u(l)$ will not expire any more.

In this algorithm the eventually timely paths guarantee agreement in the same leader process. For this reason each process resends every message it receives and this allows the rest of the processes to receive it. In the worst case, due to the periodic sending of messages in Task 1, there will be n processes sending and resending messages periodically. In terms of periodically sent messages we will have n processes sending $n-1$ messages that will be resent by the receiving $n-1$ processes, making a total of $O(n^3)$ messages sent periodically. From the point of view of links, in the worst case $O(n^2)$ links carry messages forever.

Figures 4.2 to 4.4 present three scenarios of a system composed of five processes that satisfy the assumptions required by our algorithm. Observe that, since nothing can be said about the timeliness of fair lossy links, in the presented scenarios process p_2 will eventually become the leader unless p_1 or p_5 communicate timely with p_2 through the fair lossy paths.

4.2.3 Correctness Proof

We now show the correctness of the algorithm in Figure 4.1. Let R be the set of correct processes that eventually can reach by eventually timely paths every alive process in S_1.

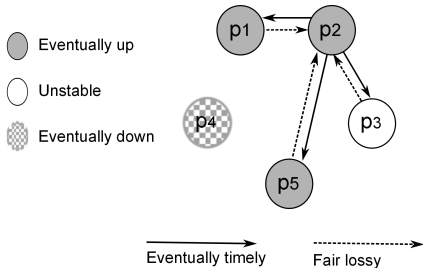

Figure 4.2: Scenario 1: three eventually up, one eventually down, one unstable.

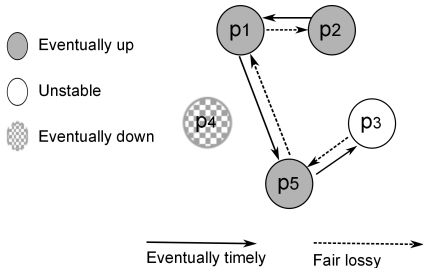

Figure 4.3: Scenario 2: three eventually up, one eventually down, one unstable.

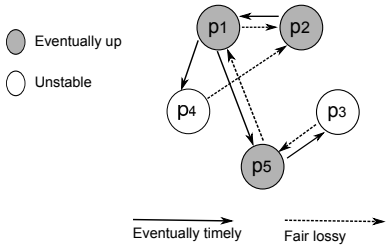

Figure 4.4: Scenario 3: three eventually up, two unstable.

By definition, there exists a constant Δ and a time GST after which every message sent by a process s, $s \in R$, takes at most Δ time to be received by every alive process. Let B be the set of correct processes p with bounded $Punish_p[p]$. As it is shown by Lemma 2, set B is not empty.

For the rest of the section we will assume that any time instant t is larger than a time t_1, where t_1 is a time instant that occurs after the stabilization time GST, and after every eventually down process has definitely crashed, and every eventually up process has definitely recovered. We will denote var_{p_t} the value of the local variable var of p at time t.

Lemma 1 *$\forall q \in correct, \forall u \in unstable, Punish_q[u]$ is unbounded.*

Proof: Consider any unstable process u. By definition, u will crash and recover an infinite number of times. Every time u recovers, it sends a ($RECOVERED$, u) message to all the processes, and hence u will send an infinite number of ($RECOVERED$,

u) messages. An infinite subset of those messages will reach some correct process q, because by definition every unstable process has at least a fair lossy link to some correct process, which will increment $Punish_q[u]$ accordingly (Line 13). Since after time t_1 correct processes will not crash, $Punish_q[u]$ will be monotonically nondecreasing and unbounded.

At any time $t > t_1$, if process $q \in R$ every message it sends will reach every correct process p in at most Δ time, setting $Punish_p[u] \geq Punish_{q_t}[u]$. If process $q \notin R$, by definition q will have at least one fair lossy path to a correct process $p \in R$, and eventually p will receive an ($ALIVE$, q, $Punish_q$) message, setting $Punish_p[u] \geq Punish_{q_t}[u]$. After that the rest of the alive processes will receive $Punish_p$ in at most $\eta + \Delta$ time. Once a correct process s receives a message from p, it will set $Punish_s[u] \geq Punish_{q_t}[u]$, and the lemma holds. ■

From now on, we will assume that any time instant t is larger than time $t_2 > t_1$, where t_2 is a time instant that occurs after every correct process $s \in R$ has $Punish_s[u]$ such that $Punish_s[u] > \eta + \Delta$.

Lemma 2 *$\forall s \in R$, $Punish_s[s]$ is bounded.*

Proof: Consider any correct process $q \neq s$. Process s sends a message ($ALIVE$, s, $Punish_s$) every η time to every process. By definition, after time GST every message that s sends is received by q within $\eta + \Delta$ time from the time q received the previous message from s. Since q increases its timer $Timeout_q[s]$ every time it expires, eventually $Timer_q(s)$ will cease expiring. Thenceforth, q will never punish s (Line 31) any more, and s will not increase $Punish_s[s]$ due to a message from any $q \in correct$.

On the other hand, every unstable process u will not reset its timers until the reception of an $ALIVE$ message from a majority of processes. Since there is a majority of correct processes, we can assure that the process u will receive a message from

at least one correct process, and u will have $Punish_u[u] \geq Punish_s[u]$ before resetting its timers. Since after time t_2, at process s, $Punish_s[u] > \eta + \Delta$, process u will have $Timeout_u[s] \geq Punish_u[u]$ (Line 17), and $Timer_u(s)$ will never expire.

Thenceforth, there is a time $t > t_2$ after which neither unstable nor correct processes will expire on s, s will not be punished (Line 31), $Punish_s[s]$ is bounded, and the lemma holds. ■

From the previous, note that $R \subseteq B$.

Lemma 3 *For every correct process $p \in B$ there exists a time after which every $q \in$ correct receives messages from p infinitely often.*

Proof: Consider a correct process $q \neq p$. We prove the contrapositive of the lemma. Suppose q does not receive messages from p infinitely often. Each time q does not receive a message from p and $Timer_q(p)$ expires, process p is punished by q in $Punish_q[p]$. Later, an infinite subset of the $ALIVE$ messages sent by q could be received by p, increasing $Punish_p[p]$, or at least by some process s, $s \in R$. The process will increase $Punish_s[p]$, and the next time p receives a message from s, it will increase $Punish_p[p]$ accordingly. If this happens infinitely often, $Punish_p[p]$ is not bounded, leading us to a contradiction. ■

For the rest of the section we will assume that any time instant t is larger than time $t_3 > t_2$, where t_3 is a time instant that occurs after $Punish_p[u] > Punish_p[p]$, $\forall u \in$ *unstable*, $\forall p \in B$, and for every eventually down process q, $q \notin Candidates_p$. This will eventually happen because $Punish_p[u]$ is unbounded and timers on every eventually down process q will expire. After that (Line 32) q will be removed from $Candidates_p$.

Lemma 4 *For every pair of correct processes p and q, $p \in B$, there is a time after which for every time t, $Punish_q[p] \geq Punish_{p_t}[p]$.*

Proof: For $q = p$, the lemma is trivial. Now assume $q \neq p$. Since $p \in B$, by Lemma 3 there exists a time after which every $q \in$ *correct* receives messages from p infinitely often. Let $t > t_3$ be any time. There is a time $t' > t$ when q receives ($ALIVE$, p, $Punish_p$), with $Punish_p[p] = c$, originally sent by p after time t, so $c \geq Punish_{p_t}[p]$. Then at time t', q sets its $Punish_q[p]$ to c, and so we have: $Punish_q[p] \geq Punish_{p_t}[p]$. The lemma now follows since $Punish_q[p]$ is monotonically nondecreasing. ■

Lemma 5 *For every correct process p:*
(1) If $Punish_p[p]$ is bounded, then there exists a value V_p and a time after which for every correct process q, $Punish_q[p] = V_p$.
(2) If $Punish_p[p]$ is not bounded, then for every correct process q, $Punish_q[p]$ is not bounded.

Proof: Let p be a correct process.

(1) Suppose $Punish_p[p]$ is bounded. Thus, by Lemma 4, for all correct processes q, there is a time $t > t_3$ after which $Punish_q[p] \geq Punish_{p_t}[p]$. Since $Punish_p[p]$ is bounded and monotonically nondecreasing, there exists a value V_p and a time after which $Punish_p[p] = V_p$. Therefore, there exists a time after which, for all correct processes q, $Punish_q[p] = V_p$.

(2) Suppose $Punish_p[p]$ is not bounded. Lemma 4 implies that $Punish_q[p]$ is also not bounded.

■

Lemma 6 *If process k is not correct then for every correct process q there is a time after which k will not be $leader_q$.*

Proof: If process k is unstable, after time $t > t_3$, $Punish_p[k] > Punish_p[p]$, for every $p \in B$. As q is correct every message broadcast (Line 15) by every process p reaches timely every correct process q, $Punish_q[k] \geq Punish_p[k]$, and process k will not be elected as leader any more. If process k is eventually down, after time t_3, $k \notin Candidates_p$. In both cases, $leader_q \neq k$ and the lemma holds. ■

Lemma 7 *There exists a correct process l and a time after which, for every correct process q, $leader_q = l$.*

Proof: Note that B is not empty. By Lemma 5(1), for every process $p \in B$, there is a corresponding integer V_p, and a time after which for every correct process q, $Punish_q[p] = V_p$ (forever). Let l denote the process p in B with the smallest corresponding tuple (V_p, p). We now show that eventually every correct process q selects l as its leader (forever). For any other process $p \neq l$: (*) there is a time after which $(Punish_q[p], p) > (Punish_q[l], l)$. This implies that eventually q selects l as its leader, forever. To show that (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 6, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $Punish_p[p]$ is bounded, then p is in B; so, by our selection of l in B, eventually $(Punish_q[p] = V_p, p) > (Punish_q[l] = V_l, l)$ forever. Finally, if $Punish_p[p]$ is not bounded, then, by Lemma 5(2), there is a time after which $Punish_q[p] > Punish_q[l] = V_l$ (because $Punish_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. It is interesting to point out that if there are two processes p and q in B with the same smallest V_p, every process will choose deterministically as its leader the same process, that will be the process with the smallest identifier between p and q. This issue is addressed by the procedure $update\_leader()$ which includes the expression *"using identifiers to break ties"*. ■

For the rest of the section we will assume that any time instant t is larger than time $t_4 > t_3$, where t_4 is a time instant that occurs after Lemma 7 holds.

Lemma 8 *There is a time after which, for every unstable process u, when up, $leader_u = \perp$ or $leader_u = l$, being l the same process as in Lemma 7.*

Proof: Every time an unstable process u recovers from a crash, it will set $leader_u$ to $\perp$. Then, u will wait until the reception of an $ALIVE$ message from a majority of processes, in order to activate its timers and call $update\_leader()$. After the waiting period, u has received a message from at least one correct process q. Once u executes Line 16, $\forall p \in S$, $Punish_u[p] \geq Punish_q[p]$, and after Line 28 $leader_u = l$. Since $l \in B$, the timers of the unstable processes will not expire on l, and the lemma holds. ■

Theorem 1 *There is a time after which (1) every correct process always trusts the same correct process l, and (2) every unstable process, when up, always trusts either $\perp$ or l. Hence, the algorithm in Figure 4.1 implements $Omega_{cr1}$ (satisfies Property 3) in system S_1.*

Proof: Follows directly from Lemma 7 and Lemma 8. ■

4.3 On the Eventual Timeliness of Fair Lossy Links

In this section we discuss an interesting behaviour associated with fair lossy links; the (eventual) *timeliness*. Let us consider a process that sends messages periodically, every η time units, through a fair lossy link. Every message sent by the process has an identifier n in the set of positive numbers. The first message has an identifier $n = 1$ and it is increased monotonically by 1 for each sending; i.e. the fourth message sent

has $n = 4$. Let us suppose that the fair lossy link delivers timely the messages whose sending identifier is even, i.e. *n mod 2 = 0*, and drops systematically the other messages.

From the definition given in Chapter 3 we know that this link is not a link of the eventually timely type because it drops messages infinitely often, but on the other hand the fair lossy link delivers messages periodically within a bounded period; i.e. it presents (eventual) *timeliness*.

Observe that due to this behaviour a correct process could become the leader even if it does not have an eventually timely path with the rest of the correct and unstable processes, provided that it can communicate timely with those processes (through fair lossy paths). If this is the case, the paths from such a process to the rest of the correct and unstable processes can be defined as *lossy but eventually timely*. Clearly, this is a behavioural definition, since *a priori* nothing can be said about the timeliness of fair lossy links. That is why we require the existence of a correct process having an eventually timely path to the rest of the correct and unstable processes, since this ensures that the algorithm stabilizes on a common and correct leader, independently of the behaviour of fair lossy links.

Figures 4.5 to 4.8 present several scenarios satisfying the assumptions required by the algorithm proposed in this chapter. We consider systems where processes p_1, p_2 and p_5 are eventually up, and processes p_3 and p_4 are unstable. In Figure 4.5, p_1 or p_2 will eventually become the leader, unless p_5 communicates timely with p_1 through the fair lossy link. In Figures 4.6 and 4.7, any of the processes p_1, p_2 or p_5 will eventually become the leader. Figure 4.8 differs from Figure 4.5 in the direct fair lossy link from p_5 to p_2. In this scenario, besides p_1 and p_2, process p_5 could also become the leader, if it can communicate timely with either p_1 or p_2.

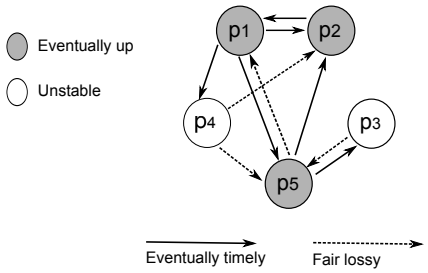

Figure 4.5: Scenario 4: three eventually up, two unstable.

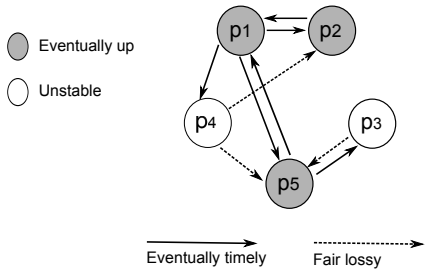

Figure 4.6: Scenario 5: three eventually up, two unstable.

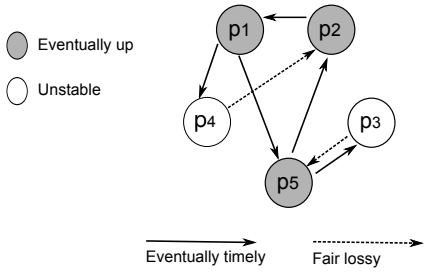

Figure 4.7: Scenario 6: three eventually up, two unstable.

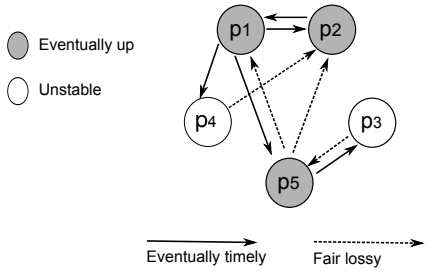

Figure 4.8: Scenario 7: three eventually up, two unstable.

Chapter 5

Omega in Crash-Recovery with Stable Storage

Contents

5.1 Introduction

In the general system model *S*, defined in Chapter 3, when a process crashes it loses the contents of all its variables. To avoid this, processes may use, if able, persistent storage, which we will call *stable storage*, whose content is preserved during crash periods. In systems where processes have access to stable storage we can implement more efficient algorithms from the point of view of communication, although there will be a cost for accessing stable storage. Access to stable storage is typically regarded as very expensive and should be minimized: i.e. the number of accesses as well as the amount of stored data should be as small as possible.

In this chapter we assume that every process has access to stable storage to keep the value of some local variables. The access to stable storage is a very important assumption because of the existence of many devices that lack of stable storage and due to the high cost associated with the writing and reading. Regarding the cost, in the algorithms that use stable storage proposed in this dissertation the access to stable storage is small, as a process only accesses stable storage a few times every time it recovers. Note that the access to stable storage is not periodical; i.e. processes access stable storage during the initialization and/or after a time, and never access it again if they do not crash and recover. In the case of the eventually up processes, after accessing stable storage in the early moments they remain up forever so they will never access it again. From this point of view, the use of stable storage in our algorithms is efficient.

The specific systems presented in this chapter also assume that every unstable process will be able to write to stable storage infinitely often. This means that unstable processes will execute the writing instructions of the algorithm infinitely often; i.e. sometimes unstable processes will crash before the writing, but other times they will not. If we suppose that unstable processes do not write to stable storage infinitely often, which is an assumption that we do not make, the algorithms will work properly and the correct processes will agree on a common leader, implementing *Omega* (Property 2).

However, unstable processes that do not satisfy the writing requirement may not agree with the rest of the unstable and correct processes, and hence the algorithm does not implement $Omega_{cr2}$. Basically, in order to agree, unstable processes must write definitely to stable storage the identity of the correct leader l.

In fact, instead of infinitely often writing it would be enough if unstable processes wrote in stable storage only one time, *after* the algorithm stabilizes and the correct processes agree on the correct leader l, but we consider that this assumption is not sufficiently general. Clearly, if an unstable process does not write the identity of the correct leader in stable storage, it cannot read the correct leader l during initialization.

Besides the use of stable storage, all the systems in this chapter share the fact that they do not require fair lossy links. For simplicity and without loss of generality, we assume that systems S_2, S_3 and S_4 are composed only of eventually timely and lossy asynchronous links. We give a brief description of these systems:

- System S_2 assumes that eventually all processes are reachable timely, i.e. through eventually timely paths, from a correct process that crashes and recovers a minimum number of times. The system also assumes unknown membership.

- System S_3 assumes that eventually all processes are reachable timely from some correct process. This system assumes known membership.

- System S_4 assumes that eventually all processes are reachable timely from some correct process, as in S_3, and that the membership is unknown, as in S_2.

In [54] Fernández et al. presented the minimal reachability conditions required to implement Omega in the crash model. The reachability conditions assumed for systems S_3 and S_4 constitute an adapted version of these. As we will see, basically, the corresponding algorithms will choose as leader the correct process that is "least suspected" among those that reach timely all processes.

The rest of the chapter is organized as follows. In Section 5.2 we present the system S_2 and give an algorithm implementing $Omega_{cr2}$ in it. In Sections 5.3 and 5.4 we present the systems S_3 and S_4, in which we weaken the synchrony assumptions and give algorithms that implement $Omega_{cr2}$ as well.

5.2 An Algorithm for System S_2

In this section we present an algorithm, adapted from [79], that implements $Omega_{cr2}$ in system S_2.

5.2.1 Specific System Assumptions in S_2

The system S_2 corresponds to the general system model S, defined in Chapter 3, with some additional assumptions. Recall that in this chapter processes have access to stable storage. In system S_2 the membership of the system – processes' identifiers – is not known *a priori* by processes. The process identifiers are totally ordered but need not be consecutive. Furthermore, processes have no knowledge about the total number of processes n in the system.

There are additional communication assumptions:

- There is an eventually timely path from one correct process c_{min}, that crashes and recovers the minimum number of times, to every correct and every unstable process.

In the case that two processes crash and recover the same number of times, the property must be satisfied by the process with the smallest identifier. Basically, this communication assumption implies that there will be a time after which the process c_{min} will have an eventually timely path, formed by correct processes that communicate through eventually timely links, to every other process. These eventually timely paths

must also reach every unstable process, although an unstable process cannot be part of the intermediate path, it can only be a final process.

5.2.2 The Algorithm

In this section we propose a distributed algorithm implementing $Omega_{cr2}$ in system S_2. Figure 5.1 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

The process chosen as leader by any process p, i.e. trusted by p, is held in a variable $leader_p$. We will show that with this algorithm there is a time after which every up process permanently has $leader_p = c_{min}$ and thus implements $Omega_{cr2}$ (satisfies Property 4).

The variable $INCARNATION_p$ contains in stable storage the number of times that the process has recovered, which we also call the *incarnation number*. Its value is 0 by default. The variable $incarnation_p$ will contain the same incarnation number, but in volatile memory. We use it to reduce the number of accesses to stable storage, by caching $INCARNATION_p$. The incarnation number of $leader_p$ is held in the variable $incarnation_{leader}$. The value of $leader_p$ is saved to stable storage in $LEADER_p$. As we will see below, the $INCARNATION_p$ and $LEADER_p$ values in stable storage constitute the foundations of the algorithm.

The basic idea of the algorithm is that eventually only process c_{min} broadcasts new $ALIVE$ messages every η time units, and that these messages reach the rest of the up processes, either directly or indirectly, by rebroadcast.

In the algorithm, when a process sends an $ALIVE$ message (Lines 14 or 19), it has necessarily incremented its incarnation number by 1 in stable storage during initialization (Line 1). Besides $incarnation_p$ every process p holds in stable storage the value of $leader_p$ (initially set to p) which is read during the execution of the initialization.

Every process p executes the following:

Initialization:
(1) increment $INCARNATION_p$ by 1 in stable storage
(2) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(3) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(4) $incarnation_{leader} \leftarrow incarnation_p$
(5) $Timeout_p \leftarrow \eta + incarnation_p$
(6) **if** $[leader_p \neq p]$ **then**
(7) reset $Timer_p$ to $Timeout_p$
(8) **end if**
(9) **start tasks** 1, 2 and 3

Task 1:
(10) wait $(\eta + incarnation_p)$
(11) write $leader_p$ to stable storage
(12) **loop forever**
(13) **if** $[leader_p = p]$ **then**
(14) broadcast $(ALIVE, p, incarnation_p)$
(15) **end if**
(16) wait (η)

Task 2:
(17) **upon reception of** message $(ALIVE, q, incarnation_q)$ with $q \neq p$
for the first time **do**
(18) **if** $[incarnation_q < incarnation_{leader}]$ **or**
$[(incarnation_q = incarnation_{leader})$ **and** $(q \leq leader_p)]$ **then**
(19) broadcast $(ALIVE, q, incarnation_q)$
(20) $leader_p \leftarrow q$
(21) $incarnation_{leader} \leftarrow incarnation_q$
(22) reset $Timer_p$ to $Timeout_p$
(23) **end if**

Task 3:
(24) **upon expiration of** $Timer_p$ **do**
(25) $Timeout_p \leftarrow Timeout_p + 1$
(26) $leader_p \leftarrow p$
(27) $incarnation_{leader} \leftarrow incarnation_p$

Figure 5.1: An algorithm implementing $Omega_{cr2}$ in S_2.

Unstable processes only have to be reachable if they remain up for a sufficiently long time. In fact, an unstable process u may crash before the reception of a message from c_{min}. This issue was taken into account when designing the algorithm.

By the assumption that every unstable process executes a write infinitely often, in Line 11 of the algorithm in Figure 5.1, we know that the *wait* in Line 10 is also executed infinitely often. The length of the *wait* is increased with every recovery since it is based on the incarnation number. Therefore, there is a time after which the *wait* will be long enough to ensure the reception of a message from c_{min} through an eventually timely path during the waiting period. This prevents every unstable process u from disagreeing because eventually and permanently $incarnation_u > incarnation_{min}$ (the incarnation number of the process c_{min}). After the *wait*, p writes the value of $leader_p$ to stable storage (Line 11). By the assumption that every unstable process is able to execute Line 11 infinitely often, eventually every unstable process will always write the correct leader to stable storage. From this point, whenever an unstable process recovers it will initially set its leader to the correct value (Line 3), implementing $Omega_{cr2}$.

Note that each process writes $leader_p$ to stable storage only once each time it starts executing the algorithm. Hence, from the point of view of the number of stable storage writing operations the algorithm is efficient. A variant of this algorithm could write this value to stable storage more frequently; e.g. periodically, or even every time it changes. This could help in speeding up the convergence at the price of a higher number of stable storage writing operations.

Removing the rebroadcast of $ALIVE$ messages (Line 19) we get a simplified version of the algorithm that works in a *fully (eventually) timely connected* system S_F; i.e. a system in which every process has a direct communication link with every other process, and all the links are eventually timely. This ensures that eventually every new $ALIVE$ message that process c_{min} broadcasts will be received timely by the rest of the up processes directly from c_{min}, and finally only process c_{min} would broadcast $ALIVE$ mes-

sages. Note that if S_F is weakened by either removing some links or considering some links as lossy asynchronous, then messages must be rebroadcast in order to guarantee their reception by all the up processes.

With regard to the cost of the algorithm in Figure 5.1: albeit eventually only c_{min} sends new messages forever, the remaining up processes will resend the message and, in the worst case, this means $n-1$ processes periodically resending messages to each of the other $n-1$ processes, making $O(n^2)$ messages sent periodically. In the worst case this implies that $O(n^2)$ links carry messages forever. If we suppose the system to be fully (eventually) timely connected we will have only one process, c_{min}, sending messages periodically and hence the cost in messages sent periodically would be $O(n)$, which is efficient.

5.2.3 Correctness Proof

This section presents the correctness proof of the algorithm in Figure 5.1.

Lemma 9 *Any message* $(ALIVE, p, incarnation_p)$, $p \in \Pi$, *eventually disappears from the system.*

Proof: Note first that a message cannot remain forever in a link, since it remains at most $GST + \delta$ time in an eventually timely link, and is lost or eventually delivered in a lossy asynchronous link. Note as well that a message cannot remain forever in a process, since by assumption processes take at least one step (execute at least one line of the algorithm) per unit of time. Then, a process will eventually crash, drop the message (Lines 17 and 18), or (re-)broadcast it (Lines 14 and 19). Finally, note that a process never rebroadcasts twice the same message and never rebroadcasts its own messages (Line 17). Hence a message can be (re-)broadcast at most n times, and will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant t is larger than a time $t_1 > t_0$, where:

(1) t_0 is a time instant that occurs after the stabilization time GST (i.e. $t_0 > GST$), and after every eventually down process has definitely crashed, every eventually up process has definitely recovered, and every unstable process has an incarnation number bigger than $incarnation_{min}$,

(2) and t_1 is a time instant such that all messages broadcast for the first time before t_0 have disappeared from the system (this eventually happens from Lemma 9). In particular, this includes (a) all messages broadcast by eventually down processes, (b) all messages broadcast by eventually up processes before recovering definitely, and (c) all messages broadcast by unstable processes with incarnation number less or equal to $incarnation_{min}$.

Lemma 10 *There is a time after which process c_{min} permanently has $leader_{c_{min}} = c_{min}$ and broadcasts a new $(ALIVE, c_{min}, incarnation_{min})$ message every η time.*

Proof: Note that after time t_1 process c_{min} will never receive an $(ALIVE, q, incarnation_q)$ message with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a process q such that $q < c_{min}$. Therefore, after time t_1 process c_{min} will never execute Lines 19-22 of the algorithm. Hence once $leader_{c_{min}} = c_{min}$ it will remain so forever. To show that this eventually happens, note that if $leader_{c_{min}} \neq c_{min}$ at time $t > t_1$, then $Timer_{c_{min}}$ must be active at that time (actually, $Timer_{c_{min}}$ was reset the last time Line 7 or 22 was executed). Since after time t_1 Lines 7 and 22 will never be executed, $Timer_{c_{min}}$ will not be reset any more. Then $Timer_{c_{min}}$ will eventually expire (Line 24), and c_{min} will set $leader_{c_{min}} = c_{min}$ and $incarnation_{leader} = incarnation_{min}$ (Lines 26-27). Finally, from Task 1, once $leader_{c_{min}} = c_{min}$, process c_{min} will permanently broadcast a

new ($ALIVE$, c_{min}, $incarnation_{min}$) message every η time. ■

Lemma 11 *There is a time after which every process $p \in correct$, $p \neq c_{min}$, permanently has either (1) $incarnation_{leader} > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_{min}$ and $leader_p \geq c_{min}$. Hence, p rebroadcasts each new ($ALIVE$, c_{min}, $incarnation_{min}$) message it receives (Line 19), since Line 18 of the algorithm will be satisfied.*

Proof: Note that after t_1, once the condition [$incarnation_{leader} > incarnation_{min}$] or [($incarnation_{leader} = incarnation_{min}$) and ($leader_p \geq c_{min}$)] is satisfied, it will remain so forever, since no ($ALIVE$, q, $incarnation_q$) message with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a process q such that $q < c_{min}$ will be received. After that, if $incarnation_{leader} < incarnation_{min}$, or if $incarnation_{leader} = incarnation_{min}$ and $leader_p < c_{min}$ at time $t > t_1$ with (1) $incarnation_p > incarnation_{min}$, or (2) $incarnation_p = incarnation_{min}$ and $p > c_{min}$, then $Timer_p$ must be active at that time. Then $Timer_p$ will eventually expire (Line 24), setting either (1) $incarnation_{leader} = incarnation_p > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_p = incarnation_{min}$ and $leader_p = p > c_{min}$. ■

Lemma 12 *There is a time after which every process $p \in correct$, $p \neq c_{min}$, permanently receives new ($ALIVE$, c_{min}, $incarnation_{min}$) messages with intervals of at most $\eta + \Delta$ time between consecutive messages, where Δ is the maximum delay introduced by an eventually timely path.*

Proof: From Lemma 10, there is a time after which c_{min} sends new messages every η time. It takes at most Δ time to a message crossing an eventually timely path from c_{min} to p. From Lemma 11 every correct process will rebroadcast every message it receives

from c_{min}. Therefore we can assure that every message sent by c_{min} will be rebroadcast by the correct process processes until the message reaches every process in the system. As c_{min} broadcasts a message every η time and the message takes at most Δ time to reach every process in the system, the lemma holds. ■

Theorem 2 *There is a time after which every up process p permanently has* $leader_p = c_{min}$*, i.e. p trusts* c_{min}*. Hence, the algorithm in Figure 5.1 implements* $Omega_{cr2}$ *(satisfies Property 4) in system* S_2.

Proof: Lemma 10 shows the claim for $p = c_{min}$. For every $p \in correct$, such that $p \neq c_{min}$, from Lemma 11 there is a time after which p permanently has either (1) $incarnation_{leader} > incarnation_{min}$, or (2) $incarnation_{leader} = incarnation_{min}$ and the variable $leader_p \geq c_{min}$. From Lemma 12, whenever $leader_p \neq c_{min}$ after this time, $leader_p$ changes back to c_{min} in at most $\eta + \Delta$ time. Furthermore, once $leader_p = c_{min}$, it only changes (to p) by executing Lines 24-27, since the conditions in Lines 17 and 18 prevent $leader_p$ from changing in Line 20. Finally, $leader_p$ changes from c_{min} to p a finite number of times, since each time this happens $Timeout_p$ is incremented by 1. By contradiction, assuming this happens an infinite number of times, $Timeout_p$ eventually grows to the point in which $Timer_p$ never expires, because new ($ALIVE$, c_{min}, $incarnation_{min}$) messages are received timely and $Timer_p$ is reset before expiration. Hence, eventually $leader_p = c_{min}$ forever. Finally, every unstable process p will eventually receive a ($ALIVE$, c_{min}, $incarnation_{min}$) message during the waiting instruction of Line 10, setting $leader_p = c_{min}$ (Line 20). Then, p will write c_{min} to stable storage (Line 11). The infinitely often writing in stable storage is one of the assumptions of the system. After that, p will have $leader_p = c_{min}$ permanently, even upon initialization (Line 6). Hence, Hence, the algorithm in Figure 5.1 implements $Omega_{cr2}$ in system S_2. ■

5.3 An Algorithm for System S_3

In this section, we propose an algorithm with a weaker synchrony assumption than the one in Section 5.2. The new system S_3 assumes that eventually all processes are reachable timely from *some* correct process, independently of its identifier and incarnation number.

The strategy followed by the algorithm is to choose as leader the correct process that is the least suspected among those that reach timely all processes. Besides this, the algorithm of this section requires the membership of the system to be known *a priori* by processes.

5.3.1 Specific System Assumptions in S_3

The system S_3 corresponds to the general system model S with some additional assumptions. Contrary to system S_2, in S_3 the membership is known. With respect to the communication assumptions, the system S_3 assumes that:

- There is an eventually timely path from some correct process to every correct and every unstable process.

Recall that in this chapter it is assumed that every process has access to stable storage, and that unstable processes are able to write to stable storage infinitely often. Finally, only eventually timely and lossy asynchronous links are considered.

5.3.2 The Algorithm

We present in this section a second algorithm, that has been adapted from [6], which implements $Omega_{cr2}$ in system S_3. Figure 5.2 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

With this algorithm there is a time after which every up process permanently has $leader_p = l$, being l the least suspected process among those that eventually communicate timely with the rest of processes.

The algorithm works as follows. Every process p has a $Punish_p[q]$ counter for every process q, which is p's estimation of the number of times q has been suspected. Process p selects as its leader the process l with the smallest $Punish_p[l]$ value. In order to keep the $Punish_p$ variable up to date, every process p broadcasts every η time units an ($ALIVE$, p, $Punish_p$) message. If a process p receives a message ($ALIVE$, q, $Punish_q$) with $q \neq p$ for the first time, p rebroadcasts the message, updates its $Punish_p$ vector accordingly, resets $Timer_p(q)$ for when it expects to receive the next ($ALIVE$, q, $Punish_q$) message, and calls the procedure $update\_leader()$.

If $Timer_p(q)$ expires before receiving a new ($ALIVE$, q, $Punish_q$) message, then p increments the suspicion counter $Punish_p[q]$, increments the value $Timeout_p[q]$, resets $Timer_p(q)$, and calls $update\_leader()$.

Unstable processes may crash before receiving (directly o indirectly) some messages from p, but they will receive messages from p infinitely often when they are up. The algorithm includes a mechanism to eventually avoid unstable processes from disturbing the leader election. This mechanism is based on the incarnation number of processes. Observe that, during initialization, every process p sets its time-outs with respect to the rest of the processes to $\eta + incarnation_p$ (Line 5). Also, p sets $Punish_p[p]$ to $incarnation_p$ (Line 8). The values set during the initialization ensure that eventually (1) every unstable process p will never suspect a correct process q that reaches timely every other process (since p's time-out with respect q keeps increasing forever, and hence eventually $Timer_p(q)$ will never expire), and consequently p will not increment $Punish_p[q]$ any more, and (2) every unstable process p will never be elected as the leader in the $update\_leader()$ procedure (since $incarnation_p$, and hence $Punish_p[p]$, keeps increasing forever).

Every process p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow l$ such that $Punish_p[l] = min\{Punish_p[q]\}$,
using identifiers to break ties

Initialization:
(2) increment $INCARNATION_p$ by 1 in stable storage
(3) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(4) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(5) $\forall q \neq p : Timeout_p[q] \leftarrow \eta + incarnation_p$
(6) $\forall q \neq p$: reset $Timer_p(q)$ to $Timeout_p[q]$
(7) $\forall q \neq p : Punish_p[q] \leftarrow 0$
(8) $Punish_p[p] \leftarrow incarnation_p$
(9) **start tasks** 1, 2 and 3

Task 1:
(10) wait $(\eta + incarnation_p)$
(11) write $leader_p$ to stable storage
(12) **loop forever**
(13) broadcast ($ALIVE$, p, $Punish_p$)
(14) wait (η)

Task 2:
(15) **upon reception of** message ($ALIVE$, q, $Punish_q$) with $q \neq p$
for the first time **do**
(16) broadcast ($ALIVE$, q, $Punish_q$)
(17) $\forall r : Punish_p[r] \leftarrow \max\{(Punish_p[r], Punish_q[r])\}$
(18) reset $Timer_p(q)$ to $Timeout_p[q]$
(19) $updateLeader()$

Task 3:
(20) **upon expiration of** $Timer_p(q)$ **do**
(21) $Punish_p[q] \leftarrow Punish_p[q] + 1$
(22) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(23) reset $Timer_p(q)$ to $Timeout_p[q]$
(24) $update\_leader()$

Figure 5.2: An algorithm implementing $Omega_{cr2}$ in S_3.

Also, the algorithm includes a waiting instruction (Line 10) followed by the writing of the leader in stable storage in order to force unstable processes to eventually agree with correct processes on the leader upon recovery.

The number of processes that send messages periodically (every η time) in this algorithm is bounded by n, the number of processes. As every process rebroadcasts the messages that receives for the first time, in the worst case we have $n-1$ processes resending $n-1$ messages to the rest of the $n-1$ processes, that make a total of $O(n^3)$ messages sent periodically. From the point of view of links that carry periodic messages, the cost is $O(n^2)$.

In the algorithm for the crash model in [6], processes (re-)broadcast explicit *ACCUSATION* messages to notify suspicions. By including the whole vector of suspicion counters into *ALIVE* messages, the algorithm in Figure 5.2 avoids the broadcast of *ACCUSATION* messages at the expense of increasing the length of the messages. Observe that the system S_3 allows scenarios in which many pairs of processes cannot communicate timely (either directly or indirectly). In [6] these processes would suspect each other and hence broadcast *ACCUSATION* messages permanently. Thus, avoiding those messages reduces notably the number of messages exchanged during the execution of the algorithm.

5.3.3 Correctness Proof

Let R be the set of correct processes that eventually reach timely all the correct and unstable processes in S_3. Let B be the set of correct processes p with bounded $Punish_p[p]$. By definition, there is a constant Δ and a time after which every message sent by s, $s \in R$, takes at most $\Delta = (n-1)(\delta + 2\sigma)$ time to be received by every correct and unstable (if up) process.

Lemma 13 *Any message* $(ALIVE, p, Punish_p)$, $p \in \Pi$, *eventually disappears from the system.*

Proof: Note first that a message cannot remain forever in a link, since it remains at most $GST + \delta$ time in an eventually timely link, and is lost or eventually delivered in a lossy asynchronous link. Note as well that a message cannot remain forever in a process, since by assumption processes take at least one step (execute at least one line of the algorithm) per unit of time. Then, a process will eventually crash, drop the message (Line 15), or (re-)broadcast it (Lines 13 or 16). Finally, note that a process never rebroadcasts twice the same message and never rebroadcasts its own messages (Line 15). Hence a message can be (re-)broadcast at most n times, and will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant t is larger than a time $t_1 > t_0$, where:

(1) t_0 is a time instant that occurs after the stabilization time GST (i.e. $t_0 > GST$), and after every eventually down process has definitely crashed, every eventually up process has definitely recovered, and every unstable process u has an incarnation number such that $incarnation_u > \Delta + 4\sigma$. Note that by definition u will crash and recover an infinite number of times, and hence eventually $incarnation_u > \Delta + 4\sigma$,

(2) and t_1 is a time instant such that all messages broadcast for the first time before t_0 have disappeared from the system (this eventually happens from Lemma 13).

Lemma 14 *$\forall s \in R$, $Punish_s[s]$ is bounded.*

Proof: Consider any correct process $q \neq s$. Process s sends a message ($ALIVE$, s, $Punish_s$) every η time. Eventually, every ($ALIVE$, s, $Punish_s$) message that s sends is received directly or indirectly by q within $\eta + \Delta$ time from the time q received the previous message from s. Since q increases $Timeout_q[s]$ every time $Timer_q(s)$ expires, eventually $Timer_q(s)$ will not expire any more. After this, q will not punish s (Line 21) again, and s will not increase $Punish_s[s]$ due to a message from any $q \in correct$.

On the other hand, every unstable process u will eventually and permanently set $Timer_u(s) > \eta + \Delta + 4\sigma$ during the initialization. Every time u resets $Timer_u(s)$, we know that $Timer_u(s)$ will expire after time $\eta + \Delta + 4\sigma$ time. As messages from s are sent every η time, in the worst case process s will send a message at time $t + \eta$, and the message will be received at process u at time $t + \eta + \Delta$, and $Timer_u(s)$ will be reset at $t + \eta + \Delta + 4\sigma$. Hence, $Timer_u(s)$ will never expire on any $s \in R$. After this, u will not punish s (Line 21) again, and s will not increase $Punish_s[s]$ due to a message from any $u \in unstable$. ■

The following observation derives from Lemma 14:

Observation 1 *$R \subseteq B$.*

Lemma 15 *For every process $p \in B$, every process $s \in R$ receives messages from p infinitely often.*

Proof: The proof is by contradiction. Assume that s does not receive messages from p infinitely often. Each time $Timer_s[p]$ expires, process p is punished by s (Line 21). Eventually, a new *ALIVE* message sent by s will be received by p and p will increase $Punish_p[p]$ (Line 17). Since this happens infinitely often, $Punish_p[p]$ is not bounded, which is a contradiction with the fact that $p \in B$. ■

The following observation derives from Lemma 15:

Observation 2 *There is a constant Δ' and a time $t_2 > t_1$ after which every message sent by $p \in B$ takes at most Δ' time to be received by every correct and unstable (if up) process.*

For the rest of the proof we will assume that any time instant t is larger than time $t_2 > t_1$, where t_2 is a time instant that occurs after $Punish_p[q] > Punish_p[p], \forall q \notin correct$

and $\forall p \in B$, and $incarnation_u > Punish_p[p]$, $\forall u \in unstable$. This will eventually happen because $Punish_p[q]$ and $incarnation_u$ grow infinitely, and by definition $Punish_p[p]$ is bounded. Note that during the initialization (Line 8) $Punish_u[u]$ is set to $incarnation_u$, so $Punish_u[u] > Punish_p[p]$.

Henceforth, var_{p_t} denotes the value of the local variable *var* of p at time t.

Lemma 16 *For every pair of correct processes p and q, $p \in B$, there is a time after which for every time t, $Punish_q[p] \geq Punish_{p_t}[p]$.*

Proof: For $p = q$, the lemma is trivial. Now assume $p \neq q$. As $p \in B$, by Lemma 15 every process $s \in R$ receives messages from p infinitely often, and hence by rebroadcast q will receive messages of type ($ALIVE$, p, $Punish_p$) infinitely often. Let $t > t_2$ be any time. There is a time $t' > t$ when q receives ($ALIVE$, p, $Punish_p$) with $Punish_p[p] = c$, originally sent by p after time t, so $c \geq Punish_{p_t}[p]$. Then at time t', q sets its $Punish_q[p]$ to c, and so we have: $Punish_q[p] \geq Punish_{p_t}[p]$. The lemma now follows since $Punish_q[p]$ is monotonically nondecreasing. ■

Lemma 17 *For every correct process p:*
(1) If $Punish_p[p]$ is bounded, then there exists a value V_p and a time after which for every correct process q, $Punish_q[p] = V_p$.
(2) If $Punish_p[p]$ is not bounded, then for every correct process q, $Punish_q[p]$ is not bounded.

Proof: Let p be a correct process.

(1) Suppose $Punish_p[p]$ is bounded. Thus, by Lemma 16, for every correct process q, there is a time $t > t_2$ after which $Punish_q[p] \geq Punish_{p_t}[p]$. Since $Punish_p[p]$ is bounded and monotonically nondecreasing, there exists a value V_p and a time after which $Punish_p[p] = V_p$. Therefore, there exists a time after which, for every correct process q, $Punish_q[p] = V_p$.

(2) Suppose $Punish_p[p]$ is not bounded. Lemma 16 implies that $Punish_q[p]$ is also not bounded.

■

Lemma 18 *For every correct process p:*
(1) If $Punish_p[p]$ is bounded, then there is a time after which for every unstable process u, $Punish_u[p] = V_p$ in at most $\Delta' + \eta + 3\sigma$ time after its initialization.
(2) If $Punish_p[p]$ is not bounded, then for every unstable process u, $Punish_u[p]$ is not bounded.

Proof: Let p be a correct process.

(1) Suppose $Punish_p[p]$ is bounded. Thus, by Lemma 17 there is a time after which $Punish_p[p] = V_p$. From Observation 2, every unstable process u will receive (if up) an alive message from every process $p \in B$ in at most $\Delta' + \eta$ time. Hence, at most $\Delta' + \eta + 3\sigma$ time after the initialization, $Punish_u[p] = V_p$.

(2) Suppose $Punish_p[p]$ is not bounded. By definition every unstable process u will receive (if up) an alive message infinitely often from every process $q \in B$, and will update $Punish_u[p]$ (Line 17). By Lemma 17, if $Punish_p[p]$ is not bounded, then $Punish_q[p]$ is not bounded. Hence, $Punish_u[p]$ is also unbounded.

■

The following observation derives from Lemma 17 and Lemma 18:

Observation 3 *There is a time $t' > t_2$ after which every message sent by every process q will contain $Punish_q[p] = V_p$, $\forall p \in B$.*

For the rest of the proof we will assume that any time instant t is larger than t' of Observation 3.

Lemma 19 *If process k is not correct then for every process q there is a time after which k will not be* $leader_q$.

Proof: As process k is not correct, there is a time $t > t_2$ after which $Punish_k[k] > Punish_p[p]$, and $Punish_p[k] > Punish_p[p]$, for every $p \in B$. If q is correct, since eventually every message broadcast by every process p reaches timely every correct process q, $Punish_q[k] \geq Punish_p[k]$, and process k will not be elected as leader any more. If q is unstable, by definition q will execute Line 11 infinitely often. By Lemma 18 there is a time after which $Punish_q[p] = V_p$ and $Punish_q[k] > Punish_q[p]$ in at most $\Delta' + \eta + 3\sigma$ time after the initialization. Hence, eventually, $leader_q \neq k$ will be permanently saved in stable storage, and process k will not be elected as leader any more. ■

Lemma 20 *There exists a correct process l and a time after which, for every correct process q,* $leader_q = l$.

Proof: Note that B is not empty. By Lemma 17(1), for every process $p \in B$, there is a corresponding integer V_p and a time after which for every correct process q, $Punish_q[p] = V_p$ (forever). Let l denote the process $p \in B$ with the smallest corresponding tuple (V_p, p). We now show that eventually every correct process q selects l as its leader (forever). For any other process $p \neq l$: (*) there is a time after which $(Punish_q[p], p) > (Punish_q[l], l)$. This implies that eventually q selects l as its leader, forever. To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 19, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $Punish_p[p]$ is bounded, then $p \in B$; so, by our selection of l in B, eventually $(Punish_q[p] = V_p, p) > (Punish_q[l] = V_l, l)$ forever. Finally, if $Punish_p[p]$ is not

bounded, then, by Lemma 17(2), there is a time after which $Punish_q[p] > Punish_q[l] = V_l$ (because $Punish_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. ■

Lemma 21 *There exists a correct process l and a time after which, for every unstable process u, $leader_u = l$.*

Proof: By Lemma 18(1), for every process $p \in B$, there is a corresponding integer V_p and a time after which for every unstable process u, $Punish_u[p] = V_p$ in $\Delta' + \eta + 3\sigma$ time after the initialization. By definition, u executes Line 11 infinitely often, saving $leader_u$ in stable storage. Let l denote the process $p \in B$ with the smallest corresponding tuple (V_p, p). We now show that eventually every unstable process u selects l as its leader (forever). For any other process $p \neq l$: (*) there is a time after which $(Punish_u[p], p) > (Punish_u[l], l)$. This implies that eventually u selects l as its leader, writes $leader_u = l$ to stable storage (forever), and reads $leader_u = l$ from stable storage during the initialization. To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 19, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $Punish_p[p]$ is bounded, then $p \in B$; so, eventually every ($ALIVE$, z, $Punish_z$) message that u receives after the initialization will contain always $(Punish_z[p] = V_p, p) > (Punish_z[l] = V_l, l)$ forever. Since during the initialization every counter is set to 0 except $Punish_u[u]$ that is unbounded, $Punish_u[p]$ will be set to $Punish_z[p]$ and $Punish_u[l]$ to $Punish_z[l]$ respectively (Line 17). By our selection of l in B, l will be chosen as leader and written in stable store at Line 11. Finally, if $Punish_p[p]$ is not bounded, then, by Lemma 18(2), there is a time after which $Punish_u[p] > Punish_u[l] = V_l$ (because $Punish_u[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. ■

Theorem 3 *There is a time after which every process that is up, either correct or unstable, always trusts the same correct process. Hence, the algorithm in Figure 5.2 implements $Omega_{cr2}$ in system S_3.*

Proof: Follows directly from Lemma 20 and Lemma 21, and the common definition of process l made in both lemmas. ■

5.4 An Algorithm for System S_4

In this section, we propose an algorithm that implements $Omega_{cr2}$ in system S_4. In this system, we weaken the assumptions in Section 5.2 by assuming unknown membership. As in S_3, we assume that eventually all processes are reachable timely from some correct process.

5.4.1 Specific System Assumptions in S_4

The system S_4 is similar to S_3 but instead of known membership, in S_4 the membership is unknown, i.e. contrary to the algorithm in Figure 5.2, the algorithm of this section does not require the membership of the system to be known *a priori* by processes. Recall that process identifiers are totally ordered, but need not be consecutive. Furthermore, processes have no knowledge about the total number of processes n. Also, every unstable process will be able to write to stable storage infinitely often. As in the previous section we have the following communication assumption:

(1) There is an eventually timely path from some correct process to every correct and every unstable process.

5.4.2 The Algorithm

We present in this section a third algorithm, adapted from [80], that implements $Omega_{cr2}$, in system S_4. Figure 5.3 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

With this algorithm there is a time after which every up process permanently has $leader_p = l$, being l the least suspected process among those that eventually communicate timely with the rest of the processes.

The algorithm works as follows. Processes send messages periodically to show they are alive. These messages are rebroadcast to attempt reaching all processes. Each process p maintains a set $Membership_p$ of pairs (q, v) (initially $(p, incarnation_p)$), where q is a process that p knows, and $v \geq 0$ is roughly the number of times that q has been "punished". Every message sent by p contains this set $Membership_p$.

When a process p receives a message from $q \neq p$ for the first time, after rebroadcasting it, for every pair $(r, -) \in Membership_q$, p checks if $(r, -) \notin Membership_p$, in which case p includes (r, v) in $Membership_p$ (being v the value associated with r in $Membership_q$), creates $Timer_p(r)$ and $Timeout_p[r]$, sets $Timeout_p[r]$ to $\eta + incarnation_p$, and resets $Timer_p(r)$. Otherwise, if $(r, -) \in Membership_p$, p updates the value associated with r in $Membership_p$. After that, p resets $Timer_p(q)$ to $Timeout_p[q]$. Then, if $(p, -) \notin Membership_q$, then p punishes itself by incrementing its associated counter in $Membership_p$. Finally, the $update\_leader()$ procedure is called to change $leader_p$ if required. A process p will hold in $leader_p$ its current leader, which is the process q whose pair (q, v) in $Membership_p$ has the smallest value v, using the process identifier to break ties.

If $Timer_p(q)$ expires before receiving a new $(ALIVE, q, Membership_q)$ message, then p increments the value associated with q in $Membership_p$, increments the value $Timeout_p[q]$, resets $Timer_p(q)$ to $Timeout_p[q]$, and calls $update\_leader()$.

Every process p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow l$ such that $v \in (l,v) = min\{v'\} \in (q,v'), \forall (q,v') \in Membership_p$
using identifiers to break ties

Initialization:
(2) increment $INCARNATION_p$ by 1 in stable storage
(3) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(4) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(5) $Membership_p \leftarrow \{(p, incarnation_p)\}$
(6) **start tasks** 1, 2 and 3

Task 1:
(7) wait $(\eta + incarnation_p)$
(8) write $leader_p$ to stable storage
(9) **loop forever**
(10) broadcast $(ALIVE, p, Membership_p)$
(11) wait (η)

Task 2:
(12) **upon reception of** message $(ALIVE, q, Membership_q)$ with $q \neq p$
for the first time **do**
(13) broadcast $(ALIVE, q, Membership_q)$
(14) $\forall (r,-) \in Membership_q$:
(15) **if** $(r,-) \notin Membership_p$ **then**
(16) $Membership_p \leftarrow Membership_p \cup \{(r,v)\} : (r,v) \in Membership_q$
(17) create $Timer_p(r)$ and $Timeout_p[r]$
(18) $Timeout_p[r] \leftarrow \eta + incarnation_p$
(19) reset $Timer_p(r)$ to $Timeout_p[r]$
(20) **else**
(21) replace in $Membership_p$
(r,v) by $(r, \max\{v,v'\}) : (r,v') \in Membership_q$
(22) **end if**
(23) reset $Timer_p(q)$ to $Timeout_p[q]$
(24) **if** $(p,-) \notin Membership_q$ **then**
(25) replace in $Membership_p$ (p,v) by $(p,v+1)$
(26) **end if**
(27) $update\_leader()$

Task 3:
(28) **upon expiration of** $Timer_p(q)$ **do**
(29) replace in $Membership_p$ (q,v) by $(q,v+1)$
(30) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(31) reset $Timer_p(q)$ to $Timeout_p[q]$
(32) $update\_leader()$

Figure 5.3: An algorithm implementing $Omega_{cr2}$ in S_4.

To avoid unstable processes from disturbing the leader election, during the initialization every process p sets $Membership_p$ with the pair $(p, incarnation_p)$ (Line 5). Also, in Task 1 p waits $\eta + incarnation_p$ units of time (Line 7) before start sending messages (that include $Membership_p$) periodically. This waiting ensures that eventually every unstable process p will only send messages with $Membership_p$ containing a pair (l, v) such that l is a correct process and v is smaller than the value associated with any other (correct or unstable) process in the system.

As the algorithm in the previous section, we have n processes sending messages periodically, and the number of messages sent periodically (every η time) is $O(n^3)$. From the point of view of links that carry messages periodically, the cost is $O(n^2)$.

In the algorithm for the crash model in [80], an additional set $candidates_p$, containing the processes considered alive, is maintained by every process p, and $ALIVE$ messages include the set $Candidates_p$ instead of $Membership_p$. Upon a suspicion on a process q, p removes q from $Candidates_p$ and broadcasts an explicit $ALIVE$ message to notify the suspicion. Again, our algorithm for the crash-recovery model avoids the explicit broadcast of messages to notify suspicions, reducing the message complexity of the algorithm.

5.4.3 Correctness Proof

Regarding the correctness proof of this algorithm, it is close to that of algorithm in Figure 5.2 that is provided in Section 5.3.3. The main differences are the unknown membership, which is addressed with a non-decreasing membership, $Membership_p$, dynamically created timers, and the punishment mechanism. By the mechanism a process p punishes itself (Lines 24-26) if it receives a message from a process r that has not received a message from p and hence, it does not contain p in $Membership_r$. This is needed because otherwise a process whose messages are always lost and hence will never be known by the rest of the processes could consider itself the leader if it is not

”punished”.

Theorem 4 *There is a time after which every process that is up, either correct or unstable, always trusts the same correct process. Hence, the algorithm in Figure 5.3 implements* $Omega_{cr2}$ *in system* S_4.

Chapter 6

Communication-Efficient Omega Algorithms

Contents

6.1 Introduction

In the algorithms presented in Chapters 4 and 5 every alive process resends messages to the rest of the processes. This is due to the fact that the connectivity assumptions are very weak (they rely on eventually timely and fair lossy paths) and hence the resending of messages is mandatory. Consequently, the cost of these algorithms is high in terms of the number of messages exchanged.

It would be desirable to have algorithms for Omega in which eventually only one process, the leader, sends messages periodically to the rest of the processes. In the system models in this dissertation, this is the minimal requirement in the number of processes that send messages periodically. Roughly speaking, the only proof we have that a process is not down permanently is the fact that at least one other process receives periodic messages from it. This thought is very significant in the crash model because processes that communicate permanently and periodically with other process are correct. In the crash-recovery model, we will need additional mechanisms to distinguish between correct and unstable processes.

In this chapter, we first define the concepts of communication efficiency and near-efficiency in the crash-recovery model in relation to the Omega failure detector. These are respectively related to the fact that eventually either only one process or correct process sends messages forever. Then we propose three algorithms that implement Omega efficiently. Specifically:

(1) A communication-efficient Omega algorithm in system S_5, where processes have access to stable storage.

(2) A near-communication-efficient Omega algorithm in a system S_6, where processes do not have access to stable storage.

(3) A communication-efficient Omega algorithm in system S_7, where there is no access to stable storage, and which relies on nondecreasing local clocks.

Recall that depending on whether or not stable storage is used, the properties that the

algorithms can satisfy varies. When stable storage is used, unstable processes can agree with correct processes by reading the identity of the leader from stable storage upon recovery. However, when stable storage is not used unstable processes must "learn" the identity of the leader from other process(es) upon recovery. It is desirable for a process to be aware of being in this learning period; e.g. to be able to inform an application querying the identity of the leader about this fact. In this regard, for system S_6 we also propose an adaptation of the near-communication-efficient algorithm that provides this capability of instability awareness and hence allows the implementation of $Omega_{cr1}$.

The rest of the chapter is organized as follows. In Section 6.2, we give the definitions of communication efficiency and near-efficiency for the Omega failure detector in crash-recovery systems. Section 6.3 presents system S_5 and a communication-efficient algorithm implementing $Omega_{cr2}$ which uses stable storage. Sections 6.4 and 6.5 present systems S_6 and S_7 in which we implement, without using stable storage, a near-communication-efficient algorithm implementing *Omega* (Property 2) and a communication-efficient algorithm implementing $Omega_{cr1}$, respectively. Finally, in Section 6.6 we discuss the relaxation of the communication reliability and synchrony assumptions.

6.2 Communication Efficiency Definitions

The system considered in this section is the general system model S presented in Chapter 3. We now define the concepts of communication-efficient and near-communication-efficient implementations of the Omega failure detector in crash-recovery models.

Definition 1 *An algorithm implementing the Omega failure detector in the crash-recovery failure model is communication-efficient if there is a time after which only one process sends messages forever.*

Definition 2 *An algorithm implementing the Omega failure detector in the crash-recovery failure model is near-communication-efficient if there is a time after which, among correct processes, only one sends messages forever.*

Intuitively, since the (correct) leader process in an Omega algorithm must send messages forever in order to continue being trusted by the rest of the processes, we can derive that a communication-efficient Omega algorithm is also near-communication-efficient. The difference between both definitions is that in a near-communication-efficient Omega algorithm unstable processes can send messages forever, as well as the leader.

In the following sections, we propose two communication-efficient Omega algorithms and a near-communication-efficient Omega algorithm.

6.3 An Algorithm for System S_5

In this section, we present a communication-efficient algorithm implementing $Omega_{cr2}$ in system S_5.

6.3.1 Specific System Assumptions in S_5

The system S_5 corresponds to the general system model S, defined in Chapter 3, with some additional assumptions.

In system S_5 we have the following communication assumption:

1) For every correct process p there is an eventually timely link from p to every correct and every unstable process.

Note that the rest of the links in S_5, i.e. the links from/to eventually down processes and the links from unstable processes, can be lossy asynchronous.

As in Chapter 5, every process has access to stable storage. Also unstable processes will write to stable storage infinitely often. Figure 6.1 presents a scenario of a system composed of five processes that meet the assumptions made in S_5.

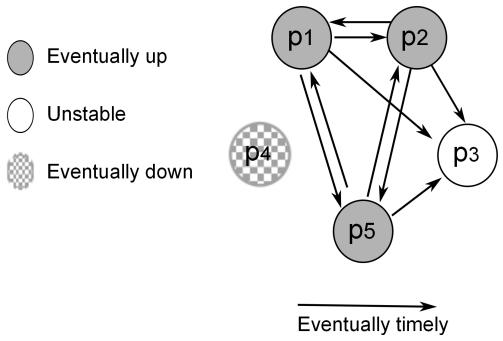

Figure 6.1: Scenario 8: three eventually up, one eventually down, one unstable.

6.3.2 The Algorithm

In this section we present a distributed algorithm that implements $Omega_{cr2}$ in system S_5. Figure 6.2 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

The process chosen as leader by a process p, i.e. trusted by p, is held in the variable $leader_p$. Every process p uses stable storage to keep the value of two local variables: $leader_p$, initially set to p; and an incarnation number $incarnation_p$, initially set to 0, which is incremented during initialization and every time p recovers from a crash. Both $incarnation_p$ and $leader_p$ are read from stable storage by p during initialization. Every process p also has a $Candidates_p$ set containing the processes from which p will choose $leader_p$ (initialized to $\{p, leader_p\}$). In addition, p has a time-out $Timeout_p[q]$ with respect to every other process q (initially set to $\eta + incarnation_p$, where η is a constant value), and a $Recovered_p$ vector to count the number of times that each process has recovered (initially set to 0 for every other process and to $incarnation_p$ for p itself).

Every process p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow$ process in $Candidates_p$ with smallest associated counter in $Recovered_p$, using identifiers to break ties

Initialization:
(2) increment $INCARNATION_p$ by 1 in stable storage
(3) $incarnation_p \leftarrow$ read $INCARNATION_p$ from stable storage
(4) $leader_p \leftarrow$ read $LEADER_p$ from stable storage
(5) $Candidates_p \leftarrow \{p, leader_p\}$
(6) **for** all $q \in \Pi$ except p:
(7) $\quad Timeout_p[q] \leftarrow \eta + incarnation_p$
(8) $\quad Recovered_p[q] \leftarrow 0$
(9) $Recovered_p[p] \leftarrow incarnation_p$
(10) **if** $leader_p \neq p$ **then**
(11) $\quad$ reset $Timer_p(leader_p)$ to $Timeout_p[leader_p]$
(12) **end if**
(13) **start tasks** 1, 2 and 3

Task 1:
(14) wait ($\eta + incarnation_p$) time units
(15) write $leader_p$ to stable storage
(16) **repeat forever every** η **time units**
(17) $\quad$ **if** $leader_p = p$ **then**
(18) $\quad\quad$ send ($LEADER$, p, $Recovered_p$) to all processes except p
(19) $\quad$ **end if**

Task 2:
(20) **upon reception of** message ($LEADER$, q, $Recovered_q$) **do**
(21) $\quad$ **for** all $r \in \Pi$:
(22) $\quad\quad Recovered_p[r] \leftarrow \max\{Recovered_p[r], Recovered_q[r]\}$
(23) $\quad$ **if** $q \notin Candidates_p$ **then**
(24) $\quad\quad Candidates_p \leftarrow Candidates_p \cup \{q\}$
(25) $\quad$ **end if**
(26) $\quad update\_leader()$
(27) $\quad$ reset $Timer_p(q)$ to $Timeout_p[q]$

Task 3:
(28) **upon expiration of** $Timer_p(q)$ **do**
(29) $\quad Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(30) $\quad Candidates_p \leftarrow Candidates_p - \{q\}$
(31) $\quad update\_leader()$

Figure 6.2: A communication-efficient $Omega_{cr2}$ algorithm in S_5.

The algorithm works as follows. After the initialization, if process p does not trust itself, it resets a timer with respect to $leader_p$. After that, p starts the three tasks of the algorithm. In Task 1, p first waits for $\eta + incarnation_p$ time units, after which it writes $leader_p$ to stable storage. Then, every η time units p checks if it trusts itself, in which case p sends a *LEADER* message containing $Recovered_p$ to the rest of the processes. Task 2 is activated whenever p receives a *LEADER* message from another process q (note that this task is active during p's waiting in Task 1): p updates $Recovered_p$ with $Recovered_q$, taking the highest value for each component of the vector. After that, p includes q in $Candidates_p$, calls the procedure $update\_leader()$ and resets $Timer_p(q)$. In the procedure $update\_leader()$, $leader_p$ is set to the process in $Candidates_p$ with the smallest associated counter in $Recovered_p$ using the processes' identifiers to brake ties. In Task 3, which is activated whenever $Timer_p(q)$ expires, p increments $Timeout_p[q]$, removes q from $Candidates_p$ and calls $update\_leader()$.

As we will show, with this algorithm eventually every correct process always trusts the same correct process ℓ. Consequently, by Task 1 eventually only one correct process sends messages forever; i.e. the algorithm is at least near-communication-efficient. With regard to the behaviour of unstable processes, the wait instruction followed by the writing of $leader_p$ to stable storage at the beginning of Task 1 ensure that eventually p will definitely write ℓ to stable storage. Recall that, as in Chapter 5, the algorithm relies on the assumption that every unstable process is able to write $leader_p$ to stable storage infinitely often. From this point on, whenever p recovers it will initialize $leader_p$ to ℓ. Moreover, the initializations of $Timeout_p[\ell]$ to $\eta + incarnation_p$ and of $Recovered_p[p]$ to $incarnation_p$ prevent unstable processes from disturbing the leader election, because they ensure that eventually: (1) every unstable process p will never suspect the leader ℓ (since p's time-out with respect to ℓ keeps increasing forever and hence eventually $Timer_p(\ell)$ will never expire); and (2) every unstable process p will never be elected as the leader in the $update\_leader()$ procedure, since $incarnation_p$,

and hence $Recovered_p[p]$, keeps increasing forever).

Finally, observe that every process only writes $leader_p$ to stable storage once every time it starts executing the algorithm; hence, the number of writings in stable storage is very low.

With regard to the cost of the algorithm in Figure 6.2, the algorithm is communication-efficient. Eventually only the leader sends messages periodically to the rest of the processes, which implies $O(n)$ messages. Furthermore, in the worst case only $O(n)$ links carry messages forever.

6.3.3 Correctness Proof

We show now that the algorithm in Figure 6.2 implements $Omega_{cr2}$ in system S_5, and that it is communication-efficient.

Lemma 22 *Any message (LEADER, p, Recovered$_p$), $p \in \Pi$, eventually disappears from the system.*

Proof: A message *m* cannot remain forever in a link, since it remains at most $GST + \delta$ time in an eventually timely link, and is lost or eventually received in a lossy asynchronous link. Also, *m* cannot remain forever in the destination process, since processes are assumed to be synchronous. Hence, *m* will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant t is larger than $t_1 > t_0$, where:

(1) t_0 is a time instant that occurs after the global stabilization time GST (i.e. $t_0 > GST$), and after every eventually down process has definitely crashed, every correct (i.e. eventually up) process has definitely recovered, and every unstable process has an incarnation value bigger than any correct process, i.e. $\forall u \in unstable$, $\forall p \in correct$: $incarnation_u > incarnation_p$,

(2) and t_1 is a time instant such that all messages sent before t_0 have disappeared from the system (this eventually happens from Lemma 22). In particular, this includes (a) all messages sent by eventually down processes, (b) all messages sent by correct processes before recovering definitely, and (c) all messages sent by every unstable process u with $Recovered_u[u] = incarnation_u \leq incarnation_p$, for every correct process p. This eventually happens, since by definition unstable processes crash and recover an infinite number of times, while correct processes crash and recover a finite number of times.

Let be ℓ the correct process with the smallest value for its $incarnation_\ell$ variable, i.e. the correct process that crashes and recovers the least times. If two or more correct processes have the same final value for their *incarnation* variables, then let ℓ be the process with the smallest identifier among them. We will show that eventually and permanently, for every correct and every unstable process p, $leader_p = \ell$.

Lemma 23 *Eventually and permanently, $leader_\ell = \ell$.*

Proof: By the algorithm, the only way for process ℓ to have as leader another process q is by receiving a message (*LEADER*, q, $Recovered_q$) such that $Recovered_q[q] < Recovered_\ell[\ell]$. However, it is simple to see that such a scenario cannot happen, since for all messages sent by q to ℓ after t, either (1) $Recovered_q[q] = incarnation_q > incarnation_\ell = Recovered_\ell[\ell]$, or (2) $Recovered_q[q] = incarnation_q = incarnation_\ell = Recovered_\ell[\ell]$ and $q > \ell$. Hence $leader_\ell$ is permanently set to ℓ in the $update\_leader()$ procedure. As a result, eventually and permanently process ℓ considers itself the leader, i.e. $leader_\ell = \ell$. ■

Lemma 24 *Eventually and permanently, process ℓ will periodically send a (LEADER, ℓ, $Recovered_\ell$) message to the rest of the processes.*

Proof: Follows directly from Lemma 23 and Task 1 of the algorithm. ■

Lemma 25 *Eventually and permanently, for every correct process p,* $leader_p = \ell$.

Proof: Follows from Lemma 23 for process ℓ. Let be any other correct process p. By Lemma 23 and Task 1 of the algorithm, ℓ will periodically send a (*LEADER*, ℓ, $Recovered_\ell$) message to the rest of the processes, including p. By the fact that the communication link between ℓ and p is eventually timely, by Task 2 p will receive the message in at most δ time units, setting $leader_p$ to ℓ in the $update\_leader()$ procedure, and resetting $Timer_p(\ell)$ to $Timeout_p[\ell]$. Observe that $Timer_p(\ell)$ can expire a finite number of times, since by Task 3 every time it expires p increments $Timeout_p[\ell]$. Hence, eventually by Task 2 p will receive a (*LEADER*, ℓ, $Recovered_\ell$) message from ℓ periodically and timely, i.e, before $Timer_p(\ell)$ expires. After this happens, p will not change $leader_p$ to a value different from ℓ any more. ■

Lemma 26 *Eventually and permanently, every correct process* $p \neq \ell$ *will not send messages any more.*

Proof: Follows directly from Lemma 25 and the algorithm. ■

Lemma 27 *Eventually, every unstable process u will not send messages any more, and* $leader_u$ *will be* ℓ *forever.*

Proof: By Lemma 23 and Task 1 of the algorithm, ℓ will periodically send a (*LEADER*, ℓ, $Recovered_\ell$) message to the rest of the processes, including u. By the facts that (1) the communication link between ℓ and u is eventually timely, and (2) u waits $\eta +$

$incarnation_u$ time units at the beginning of Task 1, eventually by Task 2 u will always receive a ($LEADER$, ℓ, $Recovered_\ell$) message from ℓ before the end of the waiting instruction of Task 1. Upon reception of that message, and since necessarily $Recovered_\ell[\ell] < Recovered_u[u]$ at process u at that instant, u adopts ℓ as its leader. Additionally, by the fact that u initializes $Timeout_u[\ell]$ to $\eta + incarnation_u$, eventually $Timer_u(\ell)$ will not expire any more. Also, at the end of the wait of Task 1, u will write ℓ to stable storage. After this happens, u will not send messages any more, and the value of $leader_u$ will be ℓ forever, since upon recovery u will read ℓ as its leader from stable storage. ■

Theorem 5 *There is a time after which every process that is up, either correct or unstable, always trusts the same correct process. The algorithm in Figure 6.2 implements $Omega_{cr2}$ in system S_5.*

Proof: Follows directly from Lemmas 23, 25 and 27. ■

Theorem 6 *The algorithm in Figure 6.2 is communication-efficient.*

Proof: Follows directly from Lemmas 24, 26 and 27. ■

6.4 An Algorithm for System S_6

In this section we present a near-communication-efficient implementation of *Omega* (satisfying Property 2) in system S_6, which assumes that processes do not have access to any form of stable storage. Remember that when a process crashes all its variables lose their values.

6.4.1 Specific System Assumptions in S_6

The system S_6 corresponds to the general system model S, defined in Chapter 3, with some additional assumptions.

The system S_6 makes the following communication assumptions:

1) For every correct process p there is an eventually timely link from p to every correct and every unstable process.

2) For every unstable process u there is a fair lossy link from u to every correct process.

The rest of the links in S_6, i.e. the links from/to eventually down processes and the links between unstable processes, can be lossy asynchronous. Figure 6.3 presents a scenario which satisfies the assumptions made in S_6.

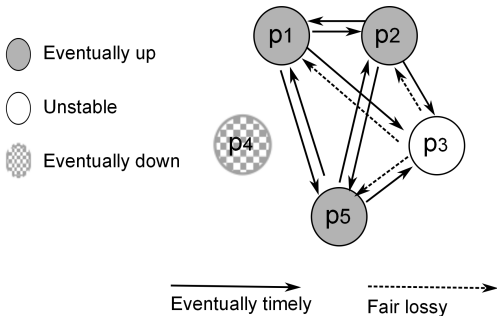

Figure 6.3: Scenario 9: three eventually up, one eventually down, one unstable.

6.4.2 The Algorithm

In this section we present an algorithm that implements Omega in system S_6. Figure 6.4 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

Contrary to the previous algorithm, where the variable $leader_p$ was initialized from stable storage, $leader_p$ is now initialized to p, as well as the set $Candidates_p$. In addition, since processes do not have an incarnation counter in stable storage, $Timeout_p[q]$ is initialized to η for every other process q, and $Recovered_p[p]$ is initialized to 1.

The algorithm works as follows. During initialization and upon recovery p sends a *RECOVERED* message to the rest of the processes in order to inform them that it has recovered. After that, p starts the three tasks of the algorithm. In Task 1, which is periodically activated every η time units, if p trusts itself then it sends a *LEADER* message containing $Recovered_p$ to the rest of the processes. Task 2 is activated whenever p receives either a *RECOVERED* message or a *LEADER* message from another process q. If p receives a *RECOVERED* message from q, p increments $Recovered_p[q]$. However, if p receives a *LEADER* message from q, p updates $Recovered_p$ with $Recovered_q$ as in the previous algorithm (i.e. taking the highest value for each component of the vector), as well as updating its time-out with respect to q ($Timeout_p[q]$), taking the higher value between its current value and that of $Recovered_p[p]$. Then p includes q in $Candidates_p$, calls the procedure $update\_leader()$ and resets $Timer_p(q)$. Task 3 remains identical to that of the previous algorithm: whenever $Timer_p(q)$ expires, p increments $Timeout_p[q]$, removes q from $Candidates_p$ and calls $update\_leader()$.

With regard to the cost of the algorithm, eventually only one correct process, the leader, sends messages periodically to the rest of the processes. However, in the worst case the rest of the processes in the system can be unstable and they could send infinitely often *RECOVERED* messages, which implies that the cost of the algorithm is $O(n^2)$ messages. Also, in the worst case $O(n^2)$ links carry messages forever.

6.4.3 Correctness Proof

We show now that the algorithm in Figure 6.4 implements *Omega* (Property 2) in system S_6 and that it is near-communication-efficient.

Every process p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow$ process in $Candidates_p$ with smallest associated counter in $Recovered_p$, using identifiers to break ties

Initialization:
(2) $leader_p \leftarrow p$
(3) $Candidates_p \leftarrow \{p\}$
(4) **for** all $q \in \Pi$ except p:
(5) $\quad Timeout_p[q] \leftarrow \eta$
(6) $\quad Recovered_p[q] \leftarrow 0$
(7) $Recovered_p[p] \leftarrow 1$
(8) send ($RECOVERED$, p) to all processes except p
(9) **start tasks** 1, 2 and 3

Task 1:
(10) **repeat forever every** η **time units**
(11) $\quad$ **if** $leader_p = p$ **then**
(12) $\quad\quad$ send ($LEADER$, p, $Recovered_p$) to all processes except p
(13) $\quad$ **end if**

Task 2:
(14) **upon reception of** message ($RECOVERED$, q)
$\quad\quad$ **or** message ($LEADER$, q, $Recovered_q$) **do**
(15) $\quad$ **if** message is of type $RECOVERED$ **then**
(16) $\quad\quad Recovered_p[q] \leftarrow Recovered_p[q] + 1$
(17) $\quad$ **else**
(18) $\quad\quad$ **for** all $r \in \Pi$:
(19) $\quad\quad\quad Recovered_p[r] \leftarrow \max\{Recovered_p[r], Recovered_q[r]\}$
(20) $\quad\quad Timeout_p[q] \leftarrow \max\{Timeout_p[q], Recovered_p[p]\}$
(21) $\quad$ **end if**
(22) $\quad$ **if** $q \notin Candidates_p$ **then**
(23) $\quad\quad Candidates_p \leftarrow Candidates_p \cup \{q\}$
(24) $\quad$ **end if**
(25) $\quad update\_leader()$
(26) $\quad$ reset $Timer_p(q)$ to $Timeout_p[q]$

Task 3:
(27) **upon expiration of** $Timer_p(q)$ **do**
(28) $\quad Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(29) $\quad Candidates_p \leftarrow Candidates_p - \{q\}$
(30) $\quad update\_leader()$

Figure 6.4: A near-communication-efficient *Omega* algorithm in S_6.

Lemma 28 *Any message m eventually disappears from the system.*

Proof: A message *m* cannot remain forever in a link, since it remains at most $GST + \delta$ time in an eventually timely link, and is lost or eventually received in a fair lossy link or a lossy asynchronous link. Also, *m* cannot remain forever in the destination process, since processes are assumed to be synchronous. Hence, *m* will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant *t* occurs after the global stabilization time *GST* (i.e. $t > GST$), and after every eventually down process has definitely crashed and disappeared forever from the set $Candidates$ of every correct and every unstable process, and every correct (i.e. eventually up) process has definitely recovered, and all $RECOVERED$ messages sent by correct processes have disappeared from the system. This eventually happens, since by definition every correct process completes the initialization of the algorithm a finite number of times, and $RECOVERED$ messages are only send during initialization.

Observation 4 $\forall p, q \in correct$: $Recovered_p[q]$ *is bounded by the number of RECOVERED messages q has sent to p.*

Lemma 29 *Eventually, no correct process will choose an unstable process as its leader.*

Proof: Let *p* and *u* be any correct process and any unstable process, respectively. There are two cases to consider:

a) Process *u* sends an infinite number of $RECOVERED$ messages to *p*. Since the communication link from *u* to *p* is fair lossy, *p* will receive an infinite number of $RECOVERED$ messages from *u*. So, eventually $Recovered_p[u] > Recovered_p[p]$ permanently, since by the algorithm $Recovered_p[p]$ is finite. After that, *p* will not choose *u* as its leader in the $update\_leader()$ procedure any more, because

$p \in Candidates_p$ permanently, and p is a better candidate than u to become the leader.

b) Process u sends a finite number of $RECOVERED$ messages to p. Since the communication link from u to p is fair lossy, this means that eventually u does not reach any more the instruction that sends the $RECOVERED$ message to p. Consequently, u will eventually disappear forever from the set $Candidates_p$. After that, p will not choose u as its leader in the $update\_leader()$ procedure any more, because $u \notin Candidates_p$ permanently.

■

Henceforth, we will consider that any time instant $t' > t$ occurs after Lemma 29 holds.

Lemma 30 *Eventually, at least one correct process p permanently has* $leader_p = p$.

Proof: By time t', eventually down processes are not in the set *Candidates* of any alive process in the system. And by Lemma 29, no correct process chooses an unstable process as its leader. Hence at every correct process p, either $leader_p = p$ and the lemma holds, or $leader_p = q$, with $q \in correct$. If p receives $LEADER$ messages periodically from any process q such that $Recovered_p[q] < Recovered_p[p]$ or $Recovered_p[q] = Recovered_p[p]$ and $q < p$, p will maintain $leader_p$ set to the process q with the minimum $Recovered_p[q]$ and the lemma holds, because this implies $leader_q = q$ at q. On the other hand, if process p does not receive such messages periodically or receives no messages at all, then p will be the process r in $Candidates_p$ with the minimum $Recovered_p[r]$, and by the algorithm eventually $leader_p = p$ and the lemma holds. ■

From the previous, we have that eventually at least one correct process p such that $leader_p = p$, will send periodically a *LEADER* message to the rest of the processes by Task 1 of the algorithm. Let K be the set of correct processes which have sent *LEADER* messages to the rest of the processes after time t. Henceforth, we will consider that any time instant $t'' > t'$ occurs after Lemma 30 holds, and after the timer on any correct process not in K (if any) has expired at every correct process.

Let $Recovered_{p_t}$ be the value of $Recovered_p$ at time t.

Lemma 31 *Eventually,* $\forall p, r \in correct$, $\forall q \in K$,
$Recovered_p[r] \leftarrow max\{Recovered_{p_t}[r], Recovered_{q_t}[r]\}$.

Proof: The only messages that process p receives from correct processes after time t are the *LEADER* messages from the processes in K. The lemma holds directly from the way $Recovered_p$ is updated in Task 2 of the algorithm. ■

Lemma 32 *Eventually,* $\forall p, q, r \in correct$, $\exists \ell \in K$,
$Recovered_q[p] = Recovered_r[p]$, *and hence* $leader_q = leader_r = leader_\ell = \ell$.

Proof: By Lemma 31 we know that $\forall q \in correct$ some correct process $\ell \in K$ will be the process in $Candidates_q$ with the smallest associated counter in $Recovered_q$. Every time $Timer_q(\ell)$ expires, by Task 3 $Timeout_q[\ell]$ is incremented. Since the link from ℓ to q is eventually timely, eventually $Timer_q(\ell)$ will not expire any more, and ℓ will remain as the leader. ■

Theorem 7 *There is a time after which every correct process always trusts the same correct process. Hence, the algorithm in Figure 6.4 implements Omega (satisfies Property 2) in system* S_6.

Proof: Follows directly from Lemma 32. ■

Theorem 8 *The algorithm in Figure 6.4 is near-communication-efficient: there is a time after which, among correct processes, only ℓ sends messages forever.*

Proof: Follows directly from Lemma 32 and Task 1 of the algorithm. ■

6.4.4 Providing Instability Awareness

Observe that in the algorithm in Figure 6.4, eventually every unstable process initially trusts itself and it could trust other (necessarily unstable) processes before trusting the leader ℓ. Hence, unstable processes could disagree with correct processes and also with each other at any time. Since this is undesirable, e.g. it could make a round fail when solving Consensus, we propose an adaptation of the algorithm that avoids this by ensuring that unstable processes do not trust any process until they trust the leader ℓ (implementing $Omega_{cr1}$).

Figure 6.5 presents in detail the adaptation, which works in system S_6 with the additional assumption that a majority of processes in the system are correct. It consists of three additional tasks, which are started by p concurrently with the rest of the tasks shown in Figure 6.4. In addition, the variable $leader_p$ is now initialized to the $\perp$ value, indicating that no process is trusted by p just after recovery.

The adaptation works as follows. In Task A, process p periodically checks if it does not trust any process, in which case p sends a *PING* message to the rest of the processes, asking for their collaboration in order to set $leader_p$ properly. Task B is activated whenever p receives a *PING* message from another process q: p replies to q with a *PONG* message that includes $Recovered_p$ and $leader_p$. Finally, Task C is activated whenever p receives a *PONG* message from another process q: if p does

not trust any process yet, then p updates $Recovered_p$ with $Recovered_q$ as usual, and includes q and $leader_q$ in $Candidates_p$. After that, if p has received so far a *PONG* message from $\lfloor \frac{n}{2} \rfloor$ different processes, then it calls the $update\_leader()$ procedure and resets a properly initialized timer on every process $r \in Candidates_p$. The check of the reception of a *PONG* message from $\lfloor \frac{n}{2} \rfloor$ different processes is also made inside the $update\_leader()$ procedure in order to keep Tasks 2 and 3 of the adapted algorithm identical to those in Figure 6.4.

Observe that the proposed adaptation does not disturb the convergence of correct processes on a common correct leader ℓ, which is carried out almost exactly as in the basic algorithm in Figure 6.4. The only difference is the initial delay until every correct process receives a *PONG* message from $\lfloor \frac{n}{2} \rfloor$ different processes, which is ensured by the existence of a majority of correct processes in the system. Eventually every correct process p stops sending *PING* messages and, consequently, p also stops receiving *PONG* messages. On the other hand, every unstable process will continue sending *PING* messages after recovery. These messages can be received by correct and unstable processes, which will reply with the corresponding *PONG* messages.

With this adapted algorithm, eventually every unstable process u will have in $leader_u$ either $\perp$, which indicates that u has not received *PONG* messages from $\lfloor \frac{n}{2} \rfloor$ different processes yet, or the common correct leader ℓ. Intuitively, after the reception of *PONG* messages from $\lfloor \frac{n}{2} \rfloor$ different processes, u will have (1) $Recovered_u$ and $Candidates_u$ such that ℓ is chosen as leader, and (2) $Timeout_u[\ell]$ such that $Timer_u(\ell)$ will never expire. After that, process u will keep ℓ as its leader until u crashes.

From the previous reasoning, we have the following theorem:

Theorem 9 *There is a time after which (1) every correct process always trusts the same correct process ℓ, and (2) every unstable process, when up, always trusts either $\perp$ or ℓ. Hence, the algorithm in Figure 6.5 implements $Omega_{cr1}$ in system S_6, assuming a majority of correct processes.*

Every process p executes the following:

procedure $update\_leader()$
(1) **if** p has received so far *PONG* from $\lfloor \frac{n}{2} \rfloor$ different processes **then**
(2) $leader_p \leftarrow$ process in $Candidates_p$ with smallest associated counter
in $Recovered_p$, using identifiers to break ties
(3) **end if**

Initialization:
(4) $leader_p \leftarrow \perp$
(5) $Candidates_p \leftarrow \{p\}$
(6) **for** all $q \in \Pi$ except p:
(7) $Timeout_p[q] \leftarrow \eta$
(8) $Recovered_p[q] \leftarrow 0$
(9) $Recovered_p[p] \leftarrow 1$
(10) send (*RECOVERED*, p) to all processes except p
(11) **start tasks** A, B, C, 1, 2 and 3

Task A:
(12) **repeat forever every** η **time units**
(13) **if** $leader_p = \perp$ **then**
(14) send (*PING*, p) to all processes except p
(15) **end if**

Task B:
(16) **upon reception of** message (*PING*, q) **do**
(17) send (*PONG*, p, $Recovered_p$, $leader_p$) to q

Task C:
(18) **upon reception of** message (*PONG*, q, $Recovered_q$, $leader_q$) **do**
(19) **if** $leader_p = \perp$ **then**
(20) **for** all $r \in \Pi$:
(21) $Recovered_p[r] \leftarrow \max\{Recovered_p[r], Recovered_q[r]\}$
(22) **if** $q \notin Candidates_p$ **then**
(23) $Candidates_p \leftarrow Candidates_p \cup \{q\}$
(24) **end if**
(25) **if** $leader_q \neq \perp$ **and** $leader_q \notin Candidates_p$ **then**
(26) $Candidates_p \leftarrow Candidates_p \cup \{leader_q\}$
(27) **end if**
(28) **if** p has received so far *PONG* from $\lfloor \frac{n}{2} \rfloor$ different processes **then**
(29) $update\_leader()$
(30) **for** all $r \in Candidates_p$ except p:
(31) $Timeout_p[r] \leftarrow \max\{Timeout_p[r], Recovered_p[p]\}$
(32) reset $Timer_p(r)$ to $Timeout_p[r]$
(33) **end if**
(34) **end if**

Task 1, Task 2 and Task 3:
(35) // same as Task 1, Task 2 and Task 3 in Figure 6.4

Figure 6.5: An algorithm implementing $Omega_{cr1}$ in extended S_6.

Finally, note that the algorithm in Figure 6.5 is not near-communication-efficient, since correct processes different from the leader send *PONG* messages forever. In the worst case, almost half of the processes in the system are unstable, and they crash and recover very often. Therefore, the cost of the algorithm from the point of view of messages exchanged periodically is $O(n^2)$. Also, in the worst case $O(n^2)$ links carry messages forever.

6.5 An Algorithm for System S_7

In this section we present the system S_7, in which we give a communication-efficient implementation of $Omega_{cr1}$ that does not rely on the use of stable storage but on a nondecreasing local clock associated with each process. With this algorithm, correct processes, i.e. those that eventually remain up forever, will eventually and permanently agree on the same correct process l. Moreover, eventually l will be the only process that keeps sending messages to the rest of the processes.

With regard to unstable processes, since stable storage is not used they must "learn" from other process(es) – actually from l – the identity of the leader upon recovery. In this regard, we make unstable processes do not trust any process upon recovery, i.e. they hold a special value $\perp$, until either they trust the leader or crash. In other words, the algorithm implements $Omega_{cr1}$.

6.5.1 Specific System Assumptions in S_7

The system S_7 corresponds to the general system model S, defined in Chapter 3, with some additional assumptions. First of all we recall that each process has access to a nondecreasing local clock that can measure time intervals with an unknown bounded drift. We assume that clocks continue running despite process crashes.

Regarding communication requirements, system S_7 makes the following assumption:

1) For every correct process p, there is an eventually timely link from p to every correct and every unstable process.

The rest of the links in S_7, i.e. the links from/to eventually down processes and the links from unstable processes, can be lossy asynchronous.

6.5.2 The Algorithm

In this section we present a communication-efficient algorithm implementing $Omega_{cr1}$ in system S_7, where processes do not have access to stable storage. Figure 6.6 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

The process chosen as leader by a process p, i.e. trusted by p, is held in the variable $leader_p$, which is initialized to the special value $\perp$, indicating that no process is trusted by p yet. Every process p also has a $Timeout_p$ variable used to set a timer with respect to its current leader, initialized to the value of the local clock, returned by the function $clock()$, as well as two timestamps, denoted by ts_p and ts_{min}, initialized to $clock()$ and ts_p respectively.

The algorithm, which is composed of three concurrent tasks that are started at the end of the initialization, works as follows. In Task 1, p first waits $Timeout_p$ time units, after which if p still has no leader, i.e. $leader_p = \perp$, then p sets $leader_p$ to p. Otherwise, p resets $Timer_p$ to $Timeout_p$ in order to monitor its current leader. Then, p enters a permanent loop in which every η time units it checks if it is the leader, i.e. $leader_p = p$, in which case p sends a ($LEADER$, p, ts_p) message to the rest of the processes.

Task 2 is activated whenever p receives a ($LEADER$, q, ts_q) message from another process q. Observe that this task is active during p's waiting instruction of Task 1. The

Every process p executes the following:

Initialization:
(1) $leader_p \leftarrow \perp$
(2) $Timeout_p \leftarrow clock()$
(3) $ts_p \leftarrow clock()$
(4) $ts_{min} \leftarrow ts_p$
(5) **start tasks** 1, 2 and 3

Task 1:
(6) wait ($Timeout_p$) time units
(7) **if** $leader_p = \perp$ **then**
(8) $leader_p \leftarrow p$
(9) **else**
(10) reset $Timer_p$ to $Timeout_p$
(11) **end if**
(12) **repeat forever every** η **time units**
(13) **if** $leader_p = p$ **then**
(14) send ($LEADER$, p, ts_p) to all processes except p
(15) **end if**

Task 2:
(16) **upon reception of** ($LEADER$, q, ts_q) **do**
(17) **if** ($ts_q < ts_{min}$)
or [($ts_q = ts_{min}$) and ($leader_p = \perp$) and ($q < p$)]
or [($ts_q = ts_{min}$) and ($leader_p \neq \perp$) and ($q \leq leader_p$)] **then**
(18) $leader_p \leftarrow q$
(19) $ts_{min} \leftarrow ts_q$
(20) reset $Timer_p$ to $Timeout_p$
(21) **end if**

Task 3:
(22) **upon expiration of** $Timer_p$ **do**
(23) $Timeout_p \leftarrow Timeout_p + 1$
(24) $leader_p \leftarrow p$
(25) $ts_{min} \leftarrow ts_p$

Figure 6.6: A communication-efficient $Omega_{cr1}$ algorithm in S_7.

received message is taken into account if either (1) $ts_q < ts_{min}$, i.e. q has recovered earlier than p's current leader, (2) ($ts_q = ts_{min}$) and ($leader_p = \perp$) and ($q < p$), i.e. p has no leader yet and q is a good candidate, or (3) ($ts_q = ts_{min}$) and ($leader_p \neq \perp$) and ($q \leq leader_p$), i.e. q is a better candidate than $leader_p$ (or $q = leader_p$). In all these cases p adopts q as its current leader, setting $leader_p$ to q and ts_{min} to ts_q, and resets $Timer_p$ to $Timeout_p$.

In Task 3, which is activated whenever $Timer_p$ expires, p "suspects" its current leader: it increments $Timeout_p$ in order to avoid premature erroneous suspicions in the future, and considers itself as the new leader, setting $leader_p$ to p and ts_{min} to ts_p.

With this algorithm, the elected leader l will be the "oldest" correct process, i.e. the process that first recovers definitely (using the process identifiers to break ties). Hence, eventually every correct process will permanently trust l. Consequently, by Task 1 eventually only one correct process will keep sending messages.

Concerning the behaviour of unstable processes, the waiting instruction at the beginning of Task 1 guarantees that, eventually and permanently, unstable processes always receive a first ($LEADER$, l, ts_l) message from l before the end of the waiting, changing their leader from $\perp$ to l by Task 2. Moreover, the initialization of $Timeout_p$ to $clock()$ prevents unstable processes from disturbing the leader election, because it ensures that eventually every unstable process u will never suspect the leader l (since u's time-out with respect l keeps increasing forever, and hence eventually $Timer_u$ will never expire). By the previous, the algorithm is communication-efficient, i.e. eventually only one process (the elected leader l) keeps sending messages forever.

With regard to the cost of the algorithm we have that eventually only the leader sends messages periodically to the rest of the processes, which implies $O(n)$ messages. Also, in the worst case only $O(n)$ links carry messages forever.

6.5.3 Correctness Proof

We show now that the algorithm in Figure 6.6 implements $Omega_{cr1}$ in system S_7 and that it is communication-efficient.

Lemma 33 *Any message (LEADER, p, ts_p), $p \in \Pi$, eventually disappears from the system.*

Proof: A message *m* cannot remain forever in a link, since it remains at most $GST + \delta$ time in an eventually timely link, and is lost or eventually received in a lossy asynchronous link. Also, *m* cannot remain forever in the destination process, since processes are assumed to be synchronous. Then, the destination process will eventually by Task 2 either take *m* into account or drop it. Hence, *m* will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant t is larger than $t_1 > t_0$, where:

(1) t_0 is a time instant that occurs after the stabilization time GST (i.e. $t_0 > GST$), and after every eventually down process has definitely crashed, every correct (i.e. eventually up) process has definitely recovered, and every unstable process has a clock value bigger than ts_p for every correct process p, i.e. $\forall u \in unstable$, $\forall p \in correct$: $ts_u > ts_p$,

(2) and t_1 is a time instant such that all messages sent before t_0 have disappeared from the system (this eventually happens from Lemma 33). In particular, this includes (a) all messages sent by eventually down processes, (b) all messages sent by correct processes before recovering definitely, and (c) all messages sent by every unstable process u with $ts_u \leq ts_p$, for every correct process p.

Let be l the correct process with the smallest value for its ts variable, i.e. the correct process that first recovers definitely. If two or more correct processes have the same final value for their ts variables, then let l be the process with the smallest identifier among them. We will show that eventually and permanently (1) for every correct process p, $leader_p = l$, and (2) for every unstable process u, either $leader_u = \perp$ or $leader_u = l$.

Lemma 34 *Eventually and permanently, $leader_l = l$.*

Proof: By the algorithm, the only way for process l to have as leader another process q is by receiving an "acceptable" message from it in Task 2. However, it is simple to see that such a scenario cannot happen, since any (*LEADER*, q, ts_q) message that l can receive necessarily has either (1) $ts_q > ts_{min} = ts_l$ at l, or (2) $ts_q = ts_{min}$ at l and $q > l$, and hence is discarded in Task 2. As a result, eventually and permanently process l considers itself the leader, i.e. $leader_l = l$. ■

Lemma 35 *Eventually and permanently, process l will periodically send a (LEADER, l, ts_l) message to the rest of the processes.*

Proof: Follows directly from Lemma 34 and the algorithm. ■

Lemma 36 *Eventually and permanently, for every correct process p, $leader_p = l$.*

Proof: Follows from Lemma 34 for process l. Let be any other correct process p. By Lemma 35 process l will periodically send a (*LEADER*, l, ts_l) message to the rest of the processes, including p. By the fact that the communication link between l and p is eventually timely, by Task 2 p will receive the message in at most δ time units, and take it into account, setting $leader_p$ to l and ts_{min} to ts_l, and resetting $Timer_p$ to $Timeout_p$.

Observe that $Timer_p$ can expire a finite number of times, since by Task 3 every time it expires p increments $Timeout_p$. Hence, eventually by Task 2 p will receive a ($LEADER$, l, ts_l) message from l periodically and timely, i.e, before $Timer_p$ expires. After this happens, p will not change $leader_p$ to a value different from l any more. ■

Lemma 37 *Eventually and permanently, every correct process $p \neq l$ will not send messages any more.*

Proof: Follows directly from Lemma 36 and the algorithm. ■

Lemma 38 *Eventually, every unstable process u will not send messages any more, and $leader_u$ will be either $\perp$ or l forever.*

Proof: By Lemma 34 and Task 1 of the algorithm, l will periodically send a ($LEADER$, l, ts_l) message to the rest of the processes, including u. By the facts that (1) the communication link between l and u is eventually timely, and (2) u waits $clock()$ time units at the beginning of Task 1, eventually by Task 2 u will always receive a ($LEADER$, l, ts_l) message from l before the end of the waiting instruction of Task 1. Upon reception of that message, and since necessarily $ts_l < ts_{min}$ at process u at that instant, u adopts l as its leader, changing the value of $leader_u$ from $\perp$ to l. In addition, by the fact that u initializes $Timeout_u$ to $clock()$, eventually $Timer_u$ will not expire any more. After this happens, u will not send messages any more. Also, the value of $leader_u$ will be either $\perp$ or l forever. ■

Theorem 10 *There is a time after which (1) every correct process always trusts the same correct process l, and (2) every unstable process, when up, always trusts either $\perp$ or l. More precisely, upon recovery an unstable process will first trust $\perp$ (i.e. it does not*

trust any process), and –if it remains up for sufficiently long– it will then trust l until it crashes. Hence, the algorithm in Figure 6.6 implements $Omega_{cr1}$ *in system* S_7.

Proof: Follows directly from Lemmas 34, 36 and 38. ■

Theorem 11 *The algorithm in Figure 6.6 is communication-efficient.*

Proof: Follows directly from Lemmas 35, 37 and 38. ■

6.6 Relaxing Communication Reliability and Synchrony

In the algorithms presented in this chapter it is possible to relax the assumptions on communication reliability and synchrony by means of message relaying; i.e. the first time a message is received, before delivering it the receiver process resends it to the rest of the processes, excluding the original sender of the message and the process from which the message has been received. This requires messages to be uniquely identified to detect duplicates. A usual way to do this in a system subject to crash failures is to add a pair (*sender_id*, *sequence_number*) to every message. In the crash-recovery failure model, uniqueness of the sequence number could be achieved using stable storage. An alternative consists of adding a timestamp provided by the sender's clock, assuming that the clocks are monotonically nondecreasing.

According to the previous, the alternative algorithms to the ones in Figure 6.2 and Figure 6.6 would work under the following weaker assumption:

1) For every correct process *p*, there is an eventually timely *path* from *p* to every correct and every unstable process.

Similarly, the alternative algorithms to the ones in Figure 6.4 and Figure 6.5 would work under the following weaker assumptions:

1) For every correct process p, there is an eventually timely *path* from p to every correct and every unstable process.

2) For every unstable process u, there is a fair lossy link from u to *some* correct process.

A consequence of the use of message relaying is that the algorithms will no longer be (near-)communication-efficient *sensu stricto*, i.e. they remain (near-)communication-efficient only regarding the number of (correct) processes that send "new" messages forever.

Chapter 7

From Omega to a $\diamond\mathcal{P}$ Failure Detector

Contents

7.1 Introduction

In this section, we address the implementation of the eventually perfect failure detector class [28], denoted by $\diamond\mathcal{P}$, in the crash-recovery model.

Our approach consists of transforming the existing implementations of the Omega failure detector, which provides eventual leader election functionality, into $\diamond\mathcal{P}$. In fact, the algorithms in Chapters 5 and 6 that implement $Omega_{cr2}$ can be used for this purpose.

The rest of the section is organized as follows. In Section 7.2 we redefine the property that $\diamond\mathcal{P}$ must satisfy in the crash-recovery model. In Section 7.3 we give the specific system assumptions. Since $\diamond\mathcal{P}$ is strictly stronger than Omega, we strengthen the system model from the previous chapters in order to make $\diamond\mathcal{P}$ implementable. Also, in Section 7.3, we propose an algorithm transforming Omega into $\diamond\mathcal{P}$, which does not require the membership of the system to be known *a priori* by processes. Finally, we propose an enhanced algorithm which provides a common set of *k* correct processes in Section 7.4.

7.2 The $\diamond\mathcal{P}$ Failure Detector in the Crash-Recovery Model

In this section we redefine the $\diamond\mathcal{P}$ failure detector for the general system model S. This definition is also valid for the system S_8 presented in Section 7.3.1. The eventually perfect failure detector $\diamond\mathcal{P}$ is the strongest *eventual* failure detector proposed by Chandra and Toueg in [28] and satisfies *strong completeness* and *eventual strong accuracy*. This basically means that there is a time after which every correct process suspects all the incorrect processes and does not suspect any correct process.

One serious drawback when implementing $\diamond\mathcal{P}$ in the crash-recovery model is that it is not possible for any process to distinguish between correct processes and unstable processes because correct processes are allowed to behave like unstable ones temporar-

ily, and it is not possible to know the instant of time after which all the correct processes remain up forever, and only the unstable processes crash and recover. Because of the existence of unstable processes in the crash-recovery model we redefine the property that $\diamond\mathcal{P}$ must satisfy:

Property 5 *There is a time after which every up process suspects every eventually down process and does not suspect any correct process.*

In order to make the algorithm and correctness proof more intelligible, we change the definition to the following:

Property 6 ($\diamond\mathcal{P}_{cr}$) *There is a time after which every up process always trusts all correct processes and does not trust any eventually down process.*

We denote by $\diamond\mathcal{P}_{cr}$ the set of failure detectors that satisfy the last property defined. There are some differences from the original definition of $\diamond\mathcal{P}$ for the crash model. First of all, the property must be satisfied by the *up* processes, which include correct processes and unstable processes, when they are up after a recovery. With regard to the trusted processes, the definition does not refer to unstable processes and thus they could be included in the trusted list infinitely often. We deal with this issue in Section 7.4.

7.3 An Algorithm Implementing $\diamond\mathcal{P}_{cr}$ in System S_8

7.3.1 Specific System Assumptions in S_8

In order to implement the transformation from Omega into $\diamond\mathcal{P}_{cr}$ in the crash-recovery model, we define the system S_8, which corresponds to the general system model S with some additional assumptions.

The membership of the system is not known, neither the total number of processes n. Processes can use stable storage and, as in Chapter 5, the algorithm relies on the

assumption that every unstable process is able to write to stable storage *infinitely often* (Line 23).

Also, we have the following communication assumptions in S_8:

1) There is an eventually timely path from some correct process p to every correct and every unstable process.

2) For every correct $q \neq p$, there is an eventually timely path from q to p.

Recall that unstable processes only must be reachable when they are up. These reachability conditions are equivalent to the minimal condition for implementing $\diamond\mathcal{P}$ in the crash model [54]. Also, these conditions together with the assumption of unique messages make lossy asynchronous links not relevant in practice, since there exists an eventually timely path between every pair of correct processes.

Finally, we assume that every process has access to a local $Omega_{cr2}$ failure detector module that satisfies Property 4. More precisely, every process p has access to a function provided by the failure detector module of $Omega_{cr2}$ (Ω_p), denoted by $\Omega_p.leader()$. This function returns the identity of the leader process trusted by the $Omega_{cr2}$ module at p at a given time (see Figure 7.1).

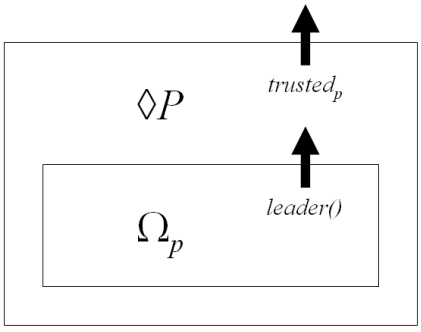

Figure 7.1: Using $Omega_{cr2}$ to build $\diamond\mathcal{P}_{cr}$.

Every process p executes the following:

Input:
Ω_p: failure detector module of $Omega_{cr2}$ at p
Output:
$Trusted_p$: set of trusted processes

Initialization:
(1) $Membership_p \leftarrow \{p\}$
(2) $Trusted_p \leftarrow$ read $TRUSTED_p$ from stable storage
(3) $wasLeader_p \leftarrow FALSE$
(4) $trustedWritten_p \leftarrow FALSE$
(5) **start tasks** 1, 2, 3 and 4

Task 1:
(6) **loop forever**
(7) **if** $[\Omega_p.leader() = p]$ **then**
(8) **if** $[wasLeader_p = FALSE]$ **then**
(9) $Trusted_p \leftarrow \{p\}$
(10) **end if**
(11) $wasLeader_p \leftarrow TRUE$
(12) broadcast $(LEADER, p, Trusted_p)$
(13) **else**
(14) $wasLeader_p \leftarrow FALSE$
(15) broadcast $(ALIVE, p)$
(16) **end if**
(17) wait (η)

Task 2:
(18) **upon reception of** message $(LEADER, q, Trusted_q)$ with $q \neq p$ for the first time **do**
(19) broadcast $(LEADER, q, Trusted_q)$
(20) **if** $[\Omega_p.leader() = q]$ **then**
(21) $Trusted_p \leftarrow Trusted_q$
(22) **if** $[trustedWritten_p = FALSE]$ **then**
(23) write $Trusted_p$ to stable storage
(24) $trustedWritten_p \leftarrow TRUE$
(25) **end if**
(26) **end if**

Task 3:
(27) **upon reception of** message $(ALIVE, q)$ with $q \neq p$ for the first time **do**
(28) broadcast $(ALIVE, q)$
(29) **if** $[\Omega_p.leader() = p]$ **then**
(30) **if** $[q \notin Membership_p]$ **then**
(31) $Membership_p \leftarrow Membership_p \cup \{q\}$
(32) create $Timer_p(q)$ and $Timeout_p[q]$
(33) $Timeout_p[q] \leftarrow \eta$
(34) **else if** $[q \notin Trusted_p]$ **then**
(35) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(36) **end if**
(37) $Trusted_p \leftarrow Trusted_p \cup \{q\}$
(38) reset $Timer_p(q)$ to $Timeout_p[q]$
(39) **end if**

Task 4:
(40) **upon expiration of** $Timer_p(q)$ **do**
(41) **if** $[q \in Trusted_p]$ **then**
(42) $Trusted_p \leftarrow Trusted_p - \{q\}$
(43) **end if**

Figure 7.2: Transforming $Omega_{cr2}$ into $\diamond\mathcal{P}_{cr}$ in S_8.

7.3.2 The Algorithm

In this section we propose a distributed algorithm that transforms $Omega_{cr2}$ into $\diamond\mathcal{P}_{cr}$ in system S_8. Figure 7.2 presents the pseudocode executed by each process when it is up. The algorithm is the collection of n instances of this pseudocode, one for each process in the system.

Every process p has two sets: a $Membership_p$ set containing all the processes p knows, initially only p, and a $Trusted_p$ set containing the processes that p trusts. When a process writes $Trusted_p$ to stable storage, it is saved to the variable $TRUSTED_P$. Every time a process p recovers during the execution of the initialization, the content of $TRUSTED_P$ is passed to $Trusted_p$.

The use of the underlying $Omega_{cr2}$ failure detector is illustrated by the calls that every process p makes to its local function $\Omega_p.leader()$ (Lines 7, 20, and 29 of the algorithm).

The approximate idea of the algorithm is the following. Every process p periodically queries its underlying $Omega_{cr2}$ module in order to know if it is the leader or not (Task 1). In case p becomes the leader (Line 8), it initializes the $Trusted_p$ set. Also, if p is the leader (either because it has become the leader, or it was the leader already), then it broadcasts a ($LEADER$, p, $Trusted_p$) message. Otherwise, i.e. if p is not the leader, it broadcasts a ($ALIVE$, p) message. These messages are rebroadcast (Lines 19 and 28) in order to allow (1) the $LEADER$ message to reach all the up processes, and (2) the $ALIVE$ messages to reach the leader. Upon reception of a ($LEADER$, q, $Trusted_q$) message, if q is the leader then p adopts $Trusted_q$ as its set of trusted processes. Also, in case p had not received previously such a message, it writes $Trusted_p$ to stable storage (Task 2). Upon reception of a ($ALIVE$, q) message, the leader knows the existence of q, increments its time-out with respect to q if required, includes q in its set of trusted processes and resets its timer with respect to q (Task 3). Finally, upon expiration of a timer, the leader removes the process whose timer has expired from its set of trusted

processes (Task 4).

With this algorithm, eventually all the up processes permanently adopt the set of trusted processes, that we will call *good_set*, periodically broadcast by the leader. And this set will eventually and permanently contain all the correct processes, and it will not contain any eventually down process.

Recall that the algorithm relies on the assumption that every unstable process is able to write to stable storage *infinitely often* (Line 23). This assumption is also required for the underlying implementation of $Omega_{cr2}$. Observe that every process writes at most once to stable storage each time it recovers when it receives the first message from the leader.

As in Chapter 5, actually, in order to agree with correct processes, it would be sufficient that every unstable process p writes at least once its $Trusted_p$ set to stable storage, provided the writing occurs after the reception of the set *good_set* from the leader.

Removing the rebroadcast of messages (Lines 19 and 28) we get a simplified version of the algorithm that works in a fully connected system; i.e. a system in which all the links are eventually timely.

Observe that the transformation algorithm in Figure 7.2 uses $Omega_{cr2}$ as a black box. If in the algorithm implementing $Omega_{cr2}$ the leader periodically broadcasts a message, e.g. to remain as leader, we could piggyback the set of trusted processes to this message, and reduce the total number of messages.

The number of processes that send messages periodically (every η time) in this algorithm is bounded by n, the number of processes. As every process rebroadcasts the messages that receives for the first time, in the worst case we have $n-1$ processes resending $O(n)$ messages to the rest of the $n-1$ processes, that make a total of $O(n^3)$ messages sent periodically. From the point of view of links that carry periodic messages, the cost is $O(n^2)$.

7.3.3 Correctness Proof

We now show the correctness of the algorithm in Figure 7.2.

Lemma 39 *Any message eventually disappears from the system.*

Proof: By definition, a message cannot remain forever in a link, since it remains at most $GST + \delta$ in an eventually timely link, and is lost or eventually delivered in a lossy asynchronous link. Note as well that a message cannot remain forever in a process, since by assumption processes take at least one step (execute at least one line of the algorithm) per unit of time. Then, a process will eventually crash, drop the message (Lines 18 and 27), or (re-)broadcast it (Lines 19 and 28). Finally, note that a process never rebroadcasts twice the same message, and never rebroadcasts its own messages (Lines 18 and 27). Hence a message can be (re-)broadcast at most n times, and will eventually disappear from the system. ■

For the rest of the proof we will assume that any time instant t is larger than a time $t'_0 > t_0$, where:

(1) t_0 is a time instant that occurs after the stabilization time GST (i.e. $t_0 > GST$), and after every eventually down process has definitely crashed, and every eventually up process has definitely recovered. Also, all messages broadcast by eventually down (up) processes for the first time before crashing (recovering) definitely have disappeared from the system (this eventually happens from Lemma 39),

(2) and t'_0 is a time instant such that the underlying $Omega_{cr2}$ algorithm has already stabilized; i.e. all the up processes always trust the same correct process *leader*.

Lemma 40 *There is a time after which process leader always trusts all correct processes, and does not trust any eventually down process.*

Proof: By definition, every *ALIVE* message broadcast by a correct process p will reach $leader$ in at most $\Delta = \delta * (n-1)$ time. Every time $Timer_{leader}(p)$ expires, p is removed from $Trusted_{leader}$ (Line 42). Since p will periodically broadcast an *ALIVE* message forever, the next time $leader$ receives an *ALIVE* message from p, $leader$ will include p in $Trusted_{leader}$ (Line 37), and will increment $Timeout_{leader}[p]$ (Line 35). Eventually $Timer_{leader}(p)$ will cease expiring (note that Δ is a bound for $Timeout_{leader}[p]$). After this, p will be permanently included in $Trusted_{leader}$. On the other hand, after time t_0 there are no *ALIVE* messages from any eventually down process q. Hence, if $leader$ has q in $Trusted_{leader}$, this implies that $leader$ has an active timer $Timer_{leader}(q)$. Since no new *ALIVE* message from q will reach $leader$, eventually $Timer_{leader}(q)$ will expire and q will be removed definitely from $Trusted_{leader}$. ■

Lemma 41 *There is a time after which every up process p always trusts all correct processes, and does not trust any eventually down process.*

Proof: To prove the lemma, we will show that eventually p agrees permanently with process $leader$. By definition, there is an eventually timely path from $leader$ to p. By Lemma 40, eventually $leader$ will broadcast a (*LEADER*, $leader$, $Trusted_{leader}$) message periodically containing all correct processes, and not containing any eventually down process (Line 12). Let p be a correct process. Eventually, p will receive these messages and will set $Trusted_p$ to $Trusted_{leader}$ forever (Line 21). On the other hand, let p be an unstable process. By the assumption that p is able to write to stable storage infinitely often (Line 23), p also sets $Trusted_p$ to $Trusted_{leader}$ infinitely often (Line 21). Hence, there is a time after which only a $Trusted_{leader}$ set containing all correct processes and not containing any eventually down process is adopted by p and written in stable storage. This set is read upon recovery by p during the execution of the initialization (Line 2). ■

Theorem 12 *The algorithm in Figure 7.2 implements $\diamond\mathcal{P}_{cr}$ (satisfies Property 6) in system S_8.*

Proof: Follows directly from Lemma 41. ■

7.4 The $\diamond\mathcal{P}_{k-cr}$ Failure Detector Class

The algorithm in Figure 7.2 allows scenarios where unstable processes are included and removed from $Trusted_{leader}$ infinitely often. In fact, $\diamond\mathcal{P}_{cr}$, which satisfies Property 6, only requires (1) all the eventually up processes to be permanently included in $Trusted_{leader}$, and (2) all the eventually down processes to be permanently excluded from $Trusted_{leader}$, i.e. it does not restrict the inclusion and removal of unstable processes. Therefore, if an eventually down process is included in $Trusted_{leader}$ it must eventually be removed forever. Since unstable processes are also incorrect, if they are included in $Trusted_{leader}$ we would also like to remove them permanently. However, it is easy to see that this is not possible, since at any given time no process can distinguish a stabilized eventually up process from an unstable but up process that will crash in the future.

Therefore, in this section we give a new definition of $\diamond\mathcal{P}$ and an algorithm that implements it. Basically, the new failure detector $\diamond\mathcal{P}_{k-cr}$ requires a parameter k, provided by the application or protocol that uses the failure detector, which is the minimum number of correct processes in the system. Note that in the worst case there is at least one correct process in the system and then $k \geq 1$.

7.4.1 Defining $\diamond\mathcal{P}_{k-cr}$

We define $\diamond\mathcal{P}_{k-cr}$ for the general system model S. This definition is also valid in system S_8. In order to implement $\diamond\mathcal{P}_{k-cr}$ we must strengthen S_8 with the additional

requirement that we assume the existence of a parameter *k*, the minimum number of correct processes in the system, that is provided to the algorithm. This number can be seen as a requirement for the application or protocol using the failure detector. Observe that in the previous section $k = 1$, but typically a higher number is usually considered, e.g., a majority. Note also that although it is not required by our algorithm, usually the application or protocol using the failure detector (e.g. Consensus), which will provide *k*, will also know the total number of processes *n*.

Assuming a known minimum number *k* of correct processes in the system, $\diamond\mathcal{P}_{k-cr}$ satisfies the following property:

Property 7 ($\diamond\mathcal{P}_{k-cr}$) *There is a time after which every up process always trusts the same set of k correct processes.*

The following observation derives from Property 7:

Observation 5

$\diamond\mathcal{P}_{1-cr} \equiv Omega_{cr2}$.

7.4.2 An Algorithm Implementing $\diamond\mathcal{P}_{k-cr}$ in System S_8

In this section we propose a distributed algorithm that transforms $Omega_{cr2}$ into $\diamond\mathcal{P}_{k-cr}$ in system S_8. Figures 7.3 and 7.4 present the pseudocode executed by each process when it is up. The algorithm is the collection of *n* instances of this pseudocode, one for each process in the system.

The algorithm is built on top of the one in Figure 7.2. Besides the $Trusted_p$ set, now every process *p* has a $corr_p$ set where *p* inserts the *k* processes that it considers correct. Basically, the leader process calculates $corr_{leader}$ using its set of trusted processes on the basis of the *incarnation number* of the processes, which is included in the *ALIVE* messages (Line 15). In the algorithms implementing $Omega_{cr2}$ presented in

Every process p executes the following:

Input:
Ω_p: failure detector module of $Omega_{cr2}$ at p
k: minimum number of correct processes in the system
Output:
$corr_p$: set of (up to k) correct, i.e. eventually up, processes

Initialization:
(1) $Membership_p \leftarrow \{p\}$
(2) read $Trusted_p$ from stable storage
(a1) read $corr_p$ from stable storage
(3) $wasLeader_p \leftarrow FALSE$
(4) $trustedWritten_p \leftarrow FALSE$
(5) **start tasks** 1, 2, 3 and 4

Task 1:
(6) **loop forever**
(7) **if** $[\Omega_p.leader() = p]$ **then**
(8) **if** $[wasLeader_p = FALSE]$ **then**
(9) $Trusted_p \leftarrow \{p\}$
(10) **end if**
(a2) $corr_p \leftarrow \{p\} \cup$ up to $k-1$ processes $\in (Trusted_p - \{p\})$ with lowest incarnation
(11) $wasLeader_p \leftarrow TRUE$
(12) broadcast ($LEADER$, p, $Trusted_p$, $corr_p$)
(13) **else**
(14) $wasLeader_p \leftarrow FALSE$
(15) broadcast ($ALIVE$, p, $\Omega_p.incarnation()$)
(16) **end if**
(17) wait (η)

Task 2:
(18) **upon reception of** message ($LEADER$, q, $Trusted_q$, $corr_q$) with $q \neq p$ for the first time **do**
(19) broadcast ($LEADER$, q, $Trusted_q$, $corr_q$)
(20) **if** $[\Omega_p.leader() = q]$ **then**
(21) $Trusted_p \leftarrow Trusted_q$
(a3) $corr_p \leftarrow corr_q$
(22) **if** $[trustedWritten_p = FALSE]$ **then**
(23) write $Trusted_p$ to stable storage
(a4) write $corr_p$ to stable storage
(24) $trustedWritten_p \leftarrow TRUE$
(25) **end if**
(26) **end if**

Figure 7.3: Transforming $Omega_{cr2}$ into $\diamond\mathcal{P}_{k-cr}$ in S_8 (Part I).

Task 3:

(27) **upon reception of** message ($ALIVE$, q, $incarnation_q$) with $q \neq p$ for the first time **do**
(28) broadcast ($ALIVE$, q, $incarnation_q$)
(29) **if** $[\Omega_p.leader() = p]$ **then**
(30) **if** $[q \notin Membership_p]$ **then**
(31) $Membership_p \leftarrow Membership_p \cup \{q\}$
(32) create $Timer_p(q)$ and $Timeout_p[q]$
(33) $Timeout_p[q] \leftarrow \eta$
(a5) create $Incarnation_p[q]$
(34) **else if** $[q \notin Trusted_p]$ **then**
(35) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(36) **end if**
(37) $Trusted_p \leftarrow Trusted_p \cup \{q\}$
(38) reset $Timer_p(q)$ to $Timeout_p[q]$
(a6) $Incarnation_p[q] \leftarrow incarnation_q$
(39) **end if**

Task 4:

(40) **upon expiration of** $Timer_p(q)$ **do**
(41) **if** $[q \in Trusted_p]$ **then**
(42) $Trusted_p \leftarrow Trusted_p - \{q\}$
(43) **end if**

Figure 7.4: Transforming $Omega_{cr2}$ into $\diamond\mathcal{P}_{k-cr}$ in S_8 (Part II).

this dissertation, every process has in stable storage a local incarnation number, whose initial value is 0, which is incremented during the execution of the initialization when it recovers from a crash. Additionally, we assume that every process p has access to a function provided by the failure detector module of $Omega_{cr2}$ at process p, denoted by $\Omega_p.incarnation()$. This function returns p's incarnation number (see Figure 7.5). Intuitively, the incarnation number of correct processes eventually stops growing, while the incarnation number of unstable processes keeps growing forever.

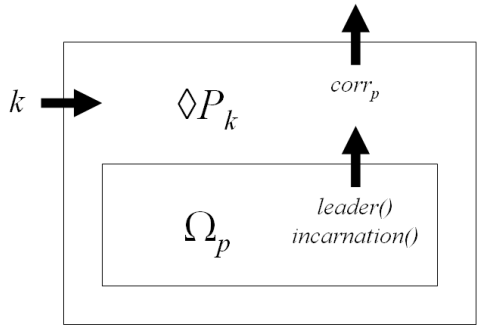

Figure 7.5: Using $Omega_{cr2}$ and the knowledge of k to build $\diamond\mathcal{P}_{k-cr}$.

Lines a1-a6 correspond to the main modifications made to the algorithm in Figure 7.2. The leader process builds the $corr_{leader}$ set by including itself and up to $k-1$ other processes in $Trusted_{leader}$ with the lowest incarnation (Line a2). This set is included in the *LEADER* message periodically broadcast. Upon reception of the *LEADER* message, the rest of the processes adopt this set and write it to stable storage if required (Lines a3-a4), as they do with the $Trusted_{leader}$ set. Also, processes read the $corr_p$ set from stable storage upon the initialization (Line a1). Finally, Lines a5-a6 correspond to the management of the processes' incarnation number by the leader.

Observe that with this algorithm, if $k = 1$, then eventually $corr_p$ will always contain just the leader process.

We use the incarnation number of the underlying algorithm implementing $Omega_{cr2}$ as an easy way to distinguish eventually up processes from unstable processes. How-

ever, we can implement $\diamond\mathcal{P}_{k-cr}$ on the base of an $Omega_{cr1}$ algorithm with a function similar to the function $incarnation()$ that provides a criteria to distinguish eventually up processes from unstable processes. For example, if we focus on the algorithm in Figure 6.4 that implements $Omega_{cr1}$ in a system where processes do not have access to stable storage, there is an array of counters with the number of times that each process has recovered, $Recovered_p$. If this implementation of $Omega_{cr1}$ would provide a function returning the value of $Recovered_p[p]$, which we can call $incarnation()$ although we know that refers to $Recovered_p[p]$, our algorithm for $\diamond\mathcal{P}_{k-cr}$ will work properly, because the value $Recovered_p[p]$ of the eventually up processes is bounded while $Recovered_p[p]$ is growing periodically in the unstable processes.

Observe that the algorithm, without stable storage, presented in Chapter 4 is not adequate in order to implement $\diamond\mathcal{P}_{k-cr}$ because the value of the array $Recovered_p[p]$ in all the correct processes, except for the leader, is not bounded.

We give now a proof sketch of the algorithm in Figures 7.3 and 7.4. Basically, the correctness follows from the proof of the algorithm in Figure 7.2 (upon which it is built), and the fact that the incarnation number of each process is incremented every time it recovers from a crash. This way, the incarnation number of correct processes eventually stops growing, while the incarnation number of unstable processes keeps growing forever. Eventually the correct leader process will have in $corr_{leader}$ the k processes with the lowest incarnation in $Trusted_{leader}$. Then, periodically the leader will send $corr_{leader}$ to the rest of the processes p that will set $corr_p$ to $corr_{leader}$. Finally, *unstable* processes will write this *good* set $corr_p$ to stable storage infinitely often, reading it upon recovery, and eventually up processes will maintain this *good* set $corr_p$ permanently.

Theorem 13 *The algorithm in Figures 7.3 and 7.4 implements $\diamond\mathcal{P}_{k-cr}$ (satisfies Property 7) in system S_8.*

Chapter 8

Aggregator Election and Data Aggregation in WSNs

Contents

8.1 Introduction

In the previous chapters we have addressed the implementation of Omega in the crash-recovery model. These algorithms, satisfying some of the properties defined in Chapter 3, provide a *leader election* service. In Chapter 7 we used this service to implement $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$. An implementation (pseudocode) of Omega can also be used as the starting point for designing a distributed algorithm, especially if part of the algorithm consists of choosing a leader.

In this chapter we propose three hierarchical aggregator election and data aggregation algorithms, based on the Omega failure detector, for large wireless sensor networks (WSNs). More precisely, the algorithms are based on those in Chapter 5. Basically, an *aggregator* is a sensor or node that collects and aggregates the sensor data generated in the WSN [112]. As the communication range of the sensors is limited, we divide the network into regions. Due to changes in the set of reachable sensors, the aggregator sensor in each region can change over time.

The algorithms in this chapter ensure that all the sensors in a region agree on a common aggregator and that all the aggregators agree on a common *super-aggregator*. Thus, the super-aggregator will collect the sensor data of the whole system regardless of the WSN size and the communication range of the sensors.

Each algorithm is implemented in a system with different connectivity assumptions. The first algorithm assumes that every pair of sensors in a region can communicate directly. The second algorithm relaxes this assumption, only requiring some correct sensor(s) to communicate directly with the rest of the sensors. Finally, the third algorithm goes a step further, by not requiring any sensor to communicate directly with the rest, but only that there is a multi-hop bidirectional path from some correct sensor(s) to the rest of the sensors.

With respect to sensor communication itself, some collision avoidance mechanisms that can be considered are TDMA, CDMA and CSMA. The main drawbacks of TDMA

and CDMA techniques are related to the requirements of sensor synchronization and a central authority to assign the time slots, and the complex modulation hardware (which is difficult to implement due to the reduced size and performance of sensors) respectively. The use of CSMA protocols such as IEEE 802.11 (WiFi), allows to minimize the energy consumption due to collisions in WSNs without requiring special capabilities and complexities of the sensors.

The rest of the chapter is organized as follows. Section 8.2 presents the related work. In Section 8.3, we describe the system model considered. Section 8.4 presents the three aggregator election algorithms for the local (intra-region) level. Section 8.5 presents the algorithm for the global (inter-region) level. Finally, in Section 8.6, we introduce a battery depletion threshold in order to enhance the quality of service (QoS) of the wireless sensor network.

8.2 Related Work

Wireless sensor networks can provide reliable data collection by applications reducing at maximum the human intervention (self-organizing and self-maintaining). Applications can act autonomously over the sensor network configuring sensors remotely, and recovering the information collected by sensors periodically or on demand. WSNs can be exposed to several sources of problems such as measuring, communication, crash and/or power supply errors. Device redundancy allows to obtain information redundancy, and then to ensure a certain level of fault tolerance. We consider a WSN with a certain clustering degree where each cluster focuses on specific areas and collects information following a distributed sensing scheme.

Wireless networked sensing applications must ensure reliable sensor data collection and aggregation, while satisfying the low-cost and low-energy operating constraints of such applications. The attempt to minimize the energy consumption leads to minimize

the amount of data transmitted by using data aggregation. Some works in the literature focus on the implementation of efficient aggregation timing control protocols as in [75]. In [123], three different data aggregation schemes —in-network, grid-based and hybrid data aggregation— are considered in order to increase the throughput, to decrease congestion and to save energy. Other works manage data aggregation with the aid of a Consensus algorithm as [83] does. The selection of the aggregator nodes is analyzed in [35], where a hierarchical energy-efficient aggregator selection protocol is presented. The protocol is probabilistic, and does not consider the failure of sensors.

Any effort aimed at extending network lifetime requires both the sensor itself and the collaborative strategy which coordinates nodes in the sensing task to be made as energy efficient as possible. Some works consider that sensor nodes can communicate directly with a base station (all proposals consider that each node can be a cluster head, which is only possible if each node can communicate with the base station) and each sensor can communicate with its neighbours present in a range of radius r [10, 74, 82, 92, 93, 109]. The number of messages transmitted to the base station should be minimized because they have a greater cost than transmission between sensor nodes. In other works, networking connectivity is powered by hopping data from sensor to sensor in search of its destination (the base station) [105]. Multihop communication in sensor networks is expected to consume less power than the traditional single hop communication [9].

Intermittent connectivity is another defining characteristic of sensor networks. Connectivity can vary continuously, as sensors can hibernate to save power, and environmental conditions can change. Intermittent connectivity causes the network to become partitioned and communication becomes unreliable. The challenge is to provide sensor reliable failure detection with the minimum number of messages.

The selection of an aggregator can be considered as a leader election, which has been extensively studied in the literature. In Chapter 2 we have commented relevant

works related to this topic.

We can find also several works focused on the hierarchization and clustering of wireless ad hoc networks, which can be easily adapted to the leader election problem as [74, 122]. In [36], a communication-efficient probabilistic quorum system is presented, which can be used for leader election. Frequent network connectivity changes are considered, possibly resulting in network partitions, and sensor crash and recovery is also considered.

Recently, MANETs have introduced a new parameter in the leader election problem: the mobility. In [73] the authors presented single-hop leader election protocols. Malpani et al. in [98] proposed two multi-hop leader election algorithms, based on [108], where any component, whose topology is static for a sufficiently long time, will eventually have exactly one leader. In [99] and [124] the algorithms overcome the previous drawbacks relying on a process majority and diffusing computations respectively.

8.3 System Model

The system model considered in this chapter is very similar to the one presented in Chapter 3. However, instead of *abstract* processes with unidirectional communication links, we consider sensors with broadcasting capabilities.

Basically, our system is a wireless sensor network where we try to collect sensor data minimizing the energy consumption. There needs to be a WSN divided into different regions where each sensor knows *a priori* its operation region and acts to transmit its sensed data to a sensor aggregator in charge of collecting all the sensed data from a specific region. Each sensor has an identifier of its operation region that is radiated in its messages. This identifier allows sensors to reject incoming messages from other regions. The operation region of a sensor, as well as the region definition, can be changed on demand. Sensor communication follows the 802.11 (WiFi) protocol.

We propose the use of an Omega algorithm to choose a common aggregator inside each region according to the reliability and battery availability of sensors. A sensor is a candidate to be elected as aggregator if it is a reliable sensor without errors during each operation period. The Omega property ensures that a unique sensor among all the candidates is elected as aggregator. The aggregator of each region is in charge of collecting all the data sensed on its region.

More formally, we consider a system S_w composed of a finite set of sensors that communicate by broadcasting messages on the wireless network. Sensors can only fail by crashing. Crashes are not permanent, i.e. crashed sensors can recover.

According to the previous, in every run and during the lifetime of its battery, we have three types of sensors:

(1) *Eventually up*. This is the subset of sensors that, after crashing and recovering a finite number of times, remain up forever, i.e. they do not crash any more. Sensors that never crash are included in this subset.

(2) *Eventually down*. This is the subset of sensors that, after crashing and recovering a finite number of times, remain down forever, i.e. they do not recover any more. Sensors that never start their execution are included in this subset.

(3) *Unstable*. This is the subset of sensors that crash and recover an infinite number of times, i.e. there is not a time after which either they remain up forever, or they remain down forever.

Once the battery of a sensor runs out, it is not considered of any particular type. Upon the hypothetical replacement of the battery, the sensor will be again a member of one of the three kinds of sensors, depending on its behaviour.

By definition, sensors in (1) are *correct*, and sensors in (2) and (3) are *incorrect*. We assume that the number of correct sensors in the system is at least one. As we will

see, correct sensors will be the candidates to become the leader, i.e. the aggregator. We also assume that every sensor has access to stable storage to keep the value of some variables.

8.3.1 Redefining the Omega Failure Detector

In Chapter 3 we have redefined the property satisfied by the Omega failure detector for the crash-recovery model. Now, we redefine the property satisfied by Omega for S_w, Property 8, considering that eventually the common leader holds until the end of its battery.

Property 8 *There is a time after which every correct sensor trusts the same correct sensor aggregator until the end of its battery.*

Accordingly, the correct sensors will trust the correct sensor aggregator until the depletion of their batteries.

8.4 Local (Intra-Region) Level

In this section, we present three aggregator election algorithms for the intra-region level:

- A first algorithm which assumes that every pair of sensors of the region can communicate directly.
- A second algorithm which assumes that some correct sensor can communicate directly with the rest of the sensors of the region.
- A third algorithm which assumes the existence of a multi-hop bidirectional path from some correct sensor to the rest of the sensors of the region.

8.4.1 A First Algorithm

We present here a first aggregator election and data aggregation algorithm for the local level, in which we assume that sensors wake up periodically to provide their sensed data, and hibernate the rest of the time. Sensor hibernation is not considered a failure, since it is a scheduled task. This assumption implies a programmed switch on/off of sensors with the aid of a clock, e.g., every $\Delta_{ACTIVATION}$ time units. We assume that hibernation periods are larger than active ones (see Figure 8.1). We assume a maximum clock skew ε between any pair of sensors.

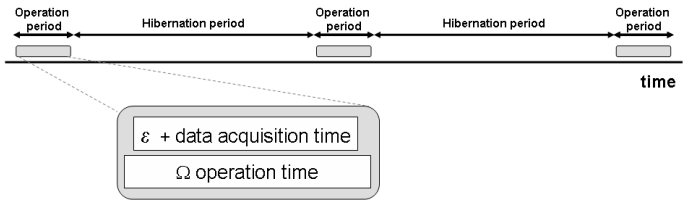

Figure 8.1: System operation time-line (Algorithm I).

The algorithm requires sensor identifiers to be totally ordered, but not necessarily consecutive. Moreover, sensors do not need to know the identifiers of the rest of the sensors in advance. We assume that all the sensors of a region can communicate directly, existing an unknown bound δ on message delay. We also assume that the execution of each line of the algorithm requires at most σ time units.

Figure 8.2 presents the pseudocode executed by each sensor when it is up. The algorithm is the collection of *n* instances of this pseudocode, one for each sensor in the system.

The algorithm uses both stable and volatile storage. The sensor chosen as aggregator by a sensor *p*, i.e. trusted by *p*, is held in the variable $leader_p$. Variables include a local incarnation number, initialized to 0, which is incremented during the execution of the initialization when a sensor recovers from a crash. Variables $incarnation_p$, $leader_p$, $incarnation_{leader}$ and $Timeout_p$ are persistently stored while variable $scheduled\_wakeup_p$

Every sensor p executes the following:

procedure $GoToHibernation()$
(1) write ($incarnation_p$, $leader_p$, $incarnation_{leader}$, $Timeout_p$) to stable storage
(2) $scheduled\_wakeup_p \leftarrow TRUE$
(3) $hibernate()$
end procedure

Initialization:
(4) read ($incarnation_p$) from stable storage
(5) **if** [$scheduled\_wakeup_p = FALSE$] **then**
(6) $incarnation_p \leftarrow incarnation_p + 1$
(7) write $incarnation_p$ to stable storage
(8) **end if**
(9) $scheduled\_wakeup_p \leftarrow FALSE$
(10) read ($leader_p$, $incarnation_{leader}$, $Timeout_p$) from stable storage
(11) **if** [$leader_p = p$] **then**
(12) **start tasks** 1 and 2
(13) **else**
(14) reset $Timer_p$ to $Timeout_p + 2\varepsilon$
(15) **start tasks** 2 and 3
(16) **end if**

Task 1:
(17) wait ε time units
(18) broadcast (I-AM-ALIVE, p, $incarnation_p$)
(19) receive $data$ from sensors during $\Delta_{DATA\_ACQUISITION}$ time
(20) $GoToHibernation()$

Task 2:
(21) **upon reception of** message (I-AM-ALIVE, q, $incarnation_q$) such that
[$incarnation_q < incarnation_{leader}$] or
[($incarnation_q = incarnation_{leader}$) and ($q \leq leader_p$)] **do**
(22) $leader_p \leftarrow q$
(23) $incarnation_{leader} \leftarrow incarnation_q$
(24) $data_p \leftarrow$ acquire sensed data
(25) send ($data_p$) to $leader_p$
(26) $GoToHibernation()$

Task 3:
(27) **upon expiration of** $Timer_p$ **do**
(28) $leader_p \leftarrow p$
(29) $incarnation_{leader} \leftarrow incarnation_p$
(30) $Timeout_p \leftarrow Timeout_p + \Delta_{TIMEOUT}$
(31) $GoToHibernation()$

Figure 8.2: Intra-region aggregator election and data aggregation (Alg. I).

remains in volatile storage. Besides this, every sensor has a local timer used to detect the potential crash of the aggregator sensor. The variable $scheduled\_wakeup_p$ keeps its value during hibernation periods, while sensor failure or battery depletion causes the lost of its value ($scheduled\_wakeup_p = FALSE$). Constant $\Delta_{TIMEOUT}$ determines the growth of the time-out in order to reach agreement. The higher this value is, the faster agreement on a common aggregator occurs. However, an excessively high value of $\Delta_{TIMEOUT}$ can induce sensors to waste their batteries, and would also delay the detection of the failure of the aggregator. Constant $\Delta_{DATA\_ACQUISITION}$ represents the maximum time passed by the aggregator during the collection of the data provided by sensors.

The algorithm starts with *Initialization* where all the values of the variables are properly recovered. After that, if the sensor considers itself as the current aggregator, the algorithm starts Tasks 1 and 2. On the other hand, if the sensor does not consider itself as the current aggregator, the algorithm resets the local timer and starts Tasks 2 and 3. Task 1 is devoted to announce the aggregator to the rest of the sensors and to collect all sensor data. Task 2, which applies to all sensors, is devoted to both send the sensor data to the aggregator, and to update the aggregator if required. Finally, Task 3 is devoted to propose the sensor itself as aggregator when the current aggregator announcement is not received before the expiration of the timer, and also increments the time-out $\Delta_{TIMEOUT}$ time units. All the tasks finish calling the *GoToHibernation()* procedure, which starts the hibernation period. Each sensor will remain in this state until the next scheduled wake-up.

Let us denote by c_{min} the correct sensor in S_w with the smallest identifier among those that have the minimum incarnation number $incarnation_{min}$. With this algorithm there is a time after which every sensor $p \in correct$ has $leader_p = c_{min}$ until the end of its battery. Eventually only sensor c_{min} broadcasts a new message (I-AM-ALIVE, c_{min}, $incarnation_{min}$) per operation period, that reaches the rest of the correct sensors.

The cost of the algorithm, measured as the number of messages sent in stability

during a data acquisition period, is linear in the number of sensors in the region $O(n)$, since the aggregator sensor broadcasts one message by Task 1, and the rest of the sensors send a message to the aggregator by Task 2.

Correctness Proof

For the rest of the section, we will assume that any time instant occurs after a time t where every eventually down sensor has definitely crashed, every eventually up sensor has definitely recovered and initialized, and every unstable sensor has an incarnation number bigger than $incarnation_{min}$. Also, all messages sent before t have already been delivered.

In order to prove the correctness of the algorithm, we formulate and prove the following lemmas and theorem. Lemma 42 proves that sensor c_{min} becomes an aggregator and notifies this fact to the rest of the sensors. Lemma 43 proves that the rest of the sensors do not declare themselves as aggregators. Lemma 44 proves that there is a time after which every correct sensor receives new (I-AM-ALIVE, c_{min}, $incarnation_{min}$) messages from c_{min}. Finally, Theorem 14 proves that the algorithm in Figure 8.2 satisfies Property 8 in system S_w.

Lemma 42 *There is a time after which sensor c_{min} permanently verifies that $leader_{c_{min}} = c_{min}$ and broadcasts a* (I-AM-ALIVE, c_{min}, $incarnation_{min}$) *message during its operation period.*

Proof: Note that after time t, sensor c_{min} will never receive a message (I-AM-ALIVE, q, $incarnation_q$) with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a sensor with identifier $q < c_{min}$. Therefore, after time t sensor c_{min} will never execute Lines 22-26 of the algorithm. Hence once $leader_{c_{min}} = c_{min}$ it will remain so until the depletion of its battery. To show that this eventually happens, note that if $leader_{c_{min}} \neq c_{min}$ at time $t' > t$, then c_{min} has $Timer_{c_{min}}$ active. Eventually, $Timer_{c_{min}}$

will expire (Line 27), setting $leader_{c_{min}} = c_{min}$ (Line 28). After that, Lines 14-15 will never be executed, since $leader_{c_{min}} = c_{min}$ holds permanently. Finally, from Task 1, once $leader_{c_{min}} = c_{min}$, sensor c_{min} will permanently broadcast a (I-AM-ALIVE, c_{min}, $incarnation_{min}$) message during its operation period (Line 18). ■

Lemma 43 *There is a time after which, every sensor $p \in correct$, $p \neq c_{min}$, permanently has either (1) $incarnation_{leader} > incarnation_{min}$, or (2) $leader_p \geq c_{min}$ and $incarnation_{leader} = incarnation_{min}$.*

Proof: Note that, after t, once [$incarnation_{leader} > incarnation_{min}$] or [($incarnation_{leader} = incarnation_{min}$) and ($leader_p \geq c_{min}$)] is satisfied, it will remain so until the depletion of the battery (either the sensor or the aggregator), since no (I-AM-ALIVE, q, $incarnation_q$) message with $incarnation_q < incarnation_{min}$, or with $incarnation_q = incarnation_{min}$ from a sensor with identifier $q < c_{min}$ will be received. Then, if $incarnation_{leader} < incarnation_{min}$, or $incarnation_{leader} = incarnation_{min}$ and $leader_p < c_{min}$ at time $t' > t$ in both cases it is satisfied that (1) $p > c_{min}$ and $incarnation_p = incarnation_{min}$, or (2) $incarnation_p > incarnation_{min}$. Then, $Timer_p$ must be active at that time, and will eventually expire (Line 27), setting (1) $incarnation_{leader} = incarnation_p > incarnation_{min}$, or (2) setting $incarnation_{leader} = incarnation_p = incarnation_{min}$ and $leader_p = p > c_{min}$ by Lines 28-29. ■

Lemma 44 *There is a time after which every sensor $p \in correct$, being $p \neq c_{min}$, permanently receives new messages* (I-AM-ALIVE, c_{min}, $incarnation_{min}$) *with intervals of at most* $(\Delta_{ACTIVATION} + \delta + 2\varepsilon + 7\sigma)$ *time between consecutive messages.*

Proof: From Lemma 42, there is a time after which c_{min} broadcasts a (I-AM-ALIVE, c_{min}, $incarnation_{min}$) message every operation period. For simplicity, let us assume that

p received the last message (I-AM-ALIVE, c_{min}, $incarnation_{min}$) from c_{min} at the beginning of the previous operation period. In the worst case, and due to clock drift, c_{min} will wake-up after almost $\Delta_{ACTIVATION} + \varepsilon$ time. When c_{min} wakes-up again, it takes $7\sigma + \varepsilon$ time to broadcast the (I-AM-ALIVE, c_{min}, $incarnation_{min}$) message. This message takes at most δ time to reach sensor p. Henceforth, the maximum time between two consecutive messages is the addition of these values $\Delta_{ACTIVATION} + \delta + 2\varepsilon + 7\sigma$. ■

Theorem 14 *There is a time after which every sensor $p \in correct$ has $leader_p = c_{min}$, i.e. p trusts c_{min}, until the end of either p's or c_{min}'s battery. Hence, the algorithm in Figure 8.2 satisfies Property 8 in system S_w.*

Proof: Lemma 42 shows the claim for $p = c_{min}$. For $p \neq c_{min}$, from Lemma 43 there is a time after which p permanently (until the end of the battery) has either (1) $incarnation_{leader} = incarnation_{min}$ and $leader_p \geq c_{min}$, or (2) $incarnation_{leader} > incarnation_{min}$. From Lemma 44, whenever $leader_p \neq c_{min}$ after this time, $leader_p$ changes to c_{min} in at most $(\Delta_{ACTIVATION} + \delta + 2\varepsilon + 7\sigma)$ time ($+2\sigma$ for the assignation of the new leader). Furthermore, once $leader_p = c_{min}$, it only changes (to p) by executing Lines 27-28, since the conditions in Line 21 prevent $leader_p$ from changing in Line 22. Finally, $leader_p$ changes from c_{min} to p a finite number of times, since each time this happens $Timeout_p$ is incremented by $\Delta_{TIMEOUT}$ time units. By contradiction, assuming this happens an infinite number of times, $Timeout_p$ eventually grows to the point in which $Timer_p$ never expires, because a new (I-AM-ALIVE, c_{min}, $incarnation_{min}$) message is received before the expiration of $Timer_p$. Hence, eventually $leader_p = c_{min}$ permanently, and thus the algorithm in Figure 8.2 satisfies Property 8 in system S_w. ■

8.4.2 A Second Algorithm

We present here a second aggregator election algorithm for the local level. Contrary to the algorithm of the previous section we assume that, in general, not every pair of sensors of a region can communicate directly. However, there exist a subset of sensors in the region that are able to reach directly the rest of the sensors of the region, and are also able to receive the messages broadcast by every sensor of the region (see Figure 8.3). Cylinders represent well-communicated sensors, i.e. candidates to become the aggregator, and circles represent sensors that cannot reach every other process in their region.

As previously, we assume that there exists an unknown bound δ on message delay, and that the execution of each line of the algorithm requires at most σ time units. In this algorithm, sensors must know the identifiers of the rest of the sensors in advance.

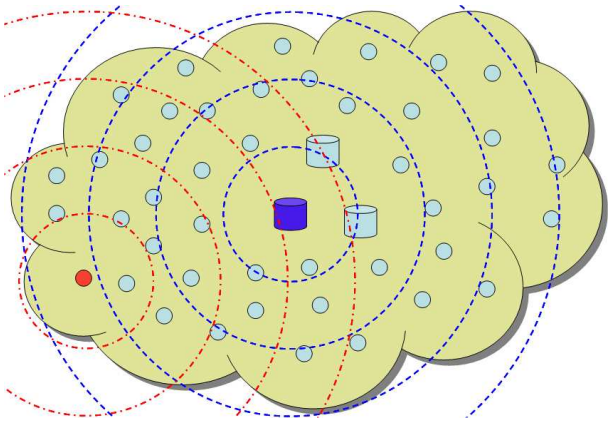

Figure 8.3: Sensor distribution in a region.

Figure 8.4 presents the pseudocode executed by each sensor when it is up. The algorithm is the collection of n instances of this pseudocode, one for each sensor in the system. With this algorithm eventually every sensor $p \in correct$ permanently has $leader_p = l$, being l the least suspected sensor among those that can communicate directly with the rest of the sensors in the region, using sensors' identifiers to break ties.

Every sensor p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow l$ such that $counter_p[l] = min\{counter_p\}$,
using identifiers to break ties
end procedure

Initialization:
(2) increment $incarnation_p$ by 1 in stable storage
(3) read ($incarnation_p$) from stable storage
(4) $\forall q \neq p : Timeout_p[q] \leftarrow \eta + incarnation_p$
(5) $\forall q \neq p$: reset $Timer_p(q)$ to $Timeout_p[q]$
(6) $\forall q \neq p : counter_p[q] \leftarrow 0$
(7) $counter_p[p] \leftarrow incarnation_p$
(8) $leader_p \leftarrow p$
(9) **start tasks** 1, 2 and 3

Task 1:
(10) **loop forever**
(11) $data_p \leftarrow$ acquire sensed data
(12) broadcast (I-AM-ALIVE, p, $counter_p$, $data_p$)
(13) wait(η)

Task 2:
(14) **upon reception of** message (I-AM-ALIVE, q, $counter_q$, $data_q$) **do**
(15) reset $Timer_p(q)$ to $Timeout_p[q]$
(16) $\forall r : counter_p[r] \leftarrow \max\{(counter_p[r], counter_q[r])\}$
(17) $update\_leader()$
(18) **if** $[leader_p = p]$ **then**
(19) collect $data_q$
(20) **end if**

Task 3:
(21) **upon expiration of** $Timer_p(q)$ **do**
(22) $counter_p[q] \leftarrow counter_p[q] + 1$
(23) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(24) reset $Timer_p(q)$ to $Timeout_p[q]$
(25) $update\_leader()$

Figure 8.4: Intra-region aggregator election and data aggregation (Alg. II).

The algorithm works as follows. Every sensor p has a $counter_p[q]$ for each sensor q, which is p's estimation of the number of times q has been suspected. Sensor p selects as its leader the sensor l with the smallest $counter_p[l]$ value. In order to acquire sensor data, and keep the $counter_p$ variable up to date, every sensor p broadcasts every η time units an (I-AM-ALIVE, p, $counter_p$, $data_p$) message, being η the interval between sensor measurements. When a sensor p receives a message (I-AM-ALIVE, q, $counter_q$, $data_q$), it resets $Timer_p(q)$ for when it expects to receive the next (I-AM-ALIVE, q, $counter_q$, $data_q$) message, updates its $counter_p$ array accordingly and calls the procedure $updateLeader()$. If p is the leader, it collects $data_q$.

If $Timer_p(q)$ expires before receiving a new (I-AM-ALIVE, q, $counter_q$, $data_q$) message, then p increments $counter_p[q]$, increments $Timeout_p[q]$, resets $Timer_p(q)$, and also calls $update\_leader()$. The following messages sent by p will include the increment of $counter_p[q]$, and this way the rest of the sensors will know about p's suspicion on q.

The algorithm includes a mechanism to eventually avoid unstable sensors from disturbing the leader election. This mechanism is based on the incarnation number of sensors. During the execution of the initialization, every sensor p initializes its time-outs with respect to the rest of the sensors to $\eta + incarnation_p$ (Line 4). Also, p initializes $counter_p[p]$ to $incarnation_p$ (Line 7). These values, set during the initialization, ensure that eventually (1) every unstable sensor p will never suspect a correct sensor q that can communicate directly with every other sensor (since p's time-out with respect q keeps increasing forever, and hence eventually $Timer_p(q)$ will never expire), and consequently p will not increment $counter_p[q]$ any more, and (2) every unstable sensor p will never be elected as the leader in the $update\_leader()$ procedure (due to the fact that $incarnation_p$, and hence $counter_p[p]$, keep increasing forever).

With regard to the cost of the algorithm in Figure 8.4, the number of messages sent during a data acquisition period (η) is linear in the number of sensors in the region $O(n)$, since every sensor broadcasts one message by Task 1.

Correctness Proof

We now show the correctness of the algorithm in Figure 8.4. For the rest of the section we will assume that any time instant occurs after a time t where every eventually down sensor has definitely crashed, every eventually up sensor has definitely recovered and initialized, and every unstable sensor u has an incarnation number such that $incarnation_u > \delta + 2\sigma$. Also, all messages sent before t have already been delivered.

Let R be the set of correct sensors that eventually can reach timely every correct sensor in S_w. Let B be the set of correct sensors p with bounded $counter_p[p]$.

Lemma 45 *$\forall s \in R$, $counter_s[s]$ is bounded.*

Proof: Consider any correct sensor $q \neq s$. Sensor s sends a message (I-AM-ALIVE, s, $counter_s$, $data_s$) every η time. By definition, every message that s sends is received by q within $\delta + \eta$ time from the time q received the previous message from s. Since q increases its timer $Timeout_q[s]$ every time it expires, eventually $Timer_q(s)$ will cease expiring. Thenceforth, q will never punish s (Line 22) any more, and s will not increase $counter_s[s]$ due to a message from any $q \in correct$.

On the other hand, every unstable sensor u will set $Timer_u(s) > \delta + \eta + 2\sigma$ during the execution of the initialization. Every time u resets $Timer_u(s)$, we know that $Timer_u(s)$ will expire after $\delta + \eta + 2\sigma$ time. As messages from s are sent every η time, in the worst case sensor s will send a message at time $t + \eta$, will be received at sensor u at time $t + \delta + \eta$, and $Timer_u(s)$ is reset at $t + \delta + \eta + 2\sigma$. Hence, $Timer_u(s)$ will never expire on any $s \in R$. Thenceforth, u will never punish s (Line 22) any more, and s will not increase $counter_s[s]$ due to a message from any $u \in unstable$. ■

From the previous, note that $R \subseteq B$.

Lemma 46 *For every correct sensor $p \in B$ there exists a time after which every sensor $q \in correct$ receives messages from p periodically.*

Proof: Consider a correct sensor $p \neq q$. We prove the contrapositive of the lemma. Suppose q does not receive messages from p periodically. Each time q does not receive a message from p and $Timer_q(p)$ expires, sensor p is punished by q in $counter_q[p]$ (Line 22). Later, the messages sent by q are received directly by p, increasing the $counter_p[p]$, or by some sensor s, $s \in R$. The sensor will increase $counter_s[p]$, and the next time p receives a message from s, it will increase $counter_p[p]$ accordingly. If this happens infinitely often, $counter_p[p]$ is not bounded, leading us to a contradiction. ■

For the rest of the section we will assume that any time instant t is larger than time $t_2 > t_1$, where t_2 is a time instant that occurs after $counter_p[q] > counter_p[p]$, $\forall q \notin correct$ and $\forall p \in B$, and $incarnation_u > counter_p[p]$, $\forall u \in unstable$. This will eventually happen because clearly $counter_p[q]$ and $incarnation_u$ grow infinitely, and by Lemma 46, $\forall p \in B$, $counter_p[p]$ is bounded. Note that during the initialization (Line 7) $counter_u[u]$ is set to $incarnation_u$, so $counter_u[u] > counter_p[p]$.

Henceforth, var_{p_t} denotes the value of the local variable *var* of p at time t.

Lemma 47 *For every pair of correct sensors p and q, $p \in B$, there is a time after which for every time t, $counter_q[p] \geq counter_{p_t}[p]$.*

Proof: For $p = q$, the lemma is trivial. Now assume $p \neq q$. As $p \in B$, by Lemma 46 there exists a time after which every $q \in correct$ receives messages from p infinitely often. Let $t > t_2$ be any time. There is a time $t' > t$ when q receives (I-AM-ALIVE, p, $counter_p$, $data_p$), with $counter_p[p] = c$, originally sent by p after time t, so $c \geq counter_{p_t}[p]$. Then at time t', q sets its $counter_q[p]$ to c, and so we have: $counter_q[p] \geq counter_{p_t}[p]$. The lemma now follows since $counter_q[p]$ is monotonically nondecreasing. ■

Lemma 48 *For every correct sensor p: 1. If $counter_p[p]$ is bounded, then there exists a value V_p and a time after which for every correct sensor q, $counter_q[p] = V_p$. 2. If*

$counter_p[p]$ is not bounded, then for every correct sensor q, $counter_q[p]$ is not bounded.

Proof: Let p be a correct sensor.

(1) Suppose $counter_p[p]$ is bounded. Thus, by Lemma 47, for every correct sensor q, there is a time $t > t_2$ after which $counter_q[p] \geq counter_{p_t}[p]$. Since $counter_p[p]$ is bounded and monotonically nondecreasing, there exists a value V_p and a time after which $counter_p[p] = V_p$. Therefore, there exists a time after which, for every correct sensor q, $counter_q[p] = V_p$.

(2) Suppose $counter_p[p]$ is not bounded. Lemma 47 implies that $counter_q[p]$ is also not bounded.

■

Lemma 49 *If sensor k is not correct then for every correct sensor q there is a time after which $leader_q \neq k$ permanently.*

Proof: As sensor k is not correct, after time $t > t_2$, $counter_p[k] > counter_p[p]$, for every $p \in B$. As q is correct every message broadcast by every sensor p reaches every correct sensor q, $counter_q[k] \geq counter_p[k]$, and sensor k will not be elected as leader any more.

■

Theorem 15 *There exists a correct sensor l and a time after which, for every correct sensor q, $leader_q = l$. Hence, the algorithm in Figure 8.4 satisfies Property 8 in system S_w.*

Proof: Note that B is not empty. By Lemma 48(1), for every sensor $p \in B$, there is a corresponding integer V_p and a time after which for every correct sensor q, $counter_q[p] = V_p$

(forever). Let l denote the sensor p in B with the smallest corresponding tuple (V_p, p). We now show that eventually every correct sensor q selects l as its leader (forever). For any other sensor $p \neq l$: (*) there is a time after which $(counter_q[p], p) > (counter_q[l], l)$. This implies that eventually q selects l as its leader, forever. To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 49, eventually p will never be elected as leader (forever). Now suppose that p is correct. If $counter_p[p]$ is bounded, then p is in B; so, by our selection of l in B, eventually $(counter_q[p] = V_p, p) > (counter_q[l] = V_l, l)$ forever. Finally, if $counter_p[p]$ is not bounded, then, by Lemma 48(2), there is a time after which $counter_q[p] > counter_q[l] = V_l$ (because $counter_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. Hence, the algorithm in Figure 8.4 satisfies Property 8 in system S_w. ■

8.4.3 A Third Algorithm

We present here a third aggregator election and data aggregation algorithm for the local level. Contrary to the previous algorithm, it does not require any sensor to communicate directly with the rest, but only the existence of a multi-hop bidirectional path from some correct sensor to the rest of the sensors. Also, similarly to the first algorithm, sensors do not need to know the identifiers of the rest of the sensors in advance. We assume that there exists an unknown bound δ on message delay, and that the execution of each line of the algorithm requires at most σ time units.

The sensor chosen as aggregator at sensor p is the sensor with the minimum associated value of $Membership_p$, denoted by $min\{Membership_p\}$, using the sensor identifier to break ties. The algorithm uses the variable $Membership_p$ to store the identifiers of the different sensors seen so far containing a set of tuples (q, v), one for each known sensor, where q is the sensor identifier and v is roughly the number of times that sensors have suspected q.

Every sensor p executes the following:

procedure $update\_leader()$
(1) $leader_p \leftarrow$ sensor in $min\{Membership_p\}$, using identifiers to break ties

end procedure

Initialization:
(2) increment $incarnation_p$ by 1 in stable storage
(3) read ($incarnation_p$) from stable storage
(4) $Membership_p \leftarrow \{(p, incarnation_p)\}$
(5) $leader_p \leftarrow p$
(6) **start tasks** 1, 2 and 3

Task 1:
(7) **loop forever**
(8) $data_p \leftarrow$ acquire sensed data
(9) broadcast (I-AM-ALIVE, p, $Membership_p$, $data_p$)
(10) wait(κ)

Task 2:
(11) **upon reception of** message (I-AM-ALIVE, q, $Membership_q$, $data_q$)
with $q \neq p$ for the first time **do**
(12) broadcast (I-AM-ALIVE, q, $Membership_q$, $data_q$)
(13) $\forall (r,-) \in Membership_q$:
(14) **if** $(r,-) \notin Membership_p$ **then**
(15) $Membership_p \leftarrow Membership_p \cup \{(r,v)\} : (r,v) \in Membership_q$
(16) create $Timer_p(r)$ and $Timeout_p[r]$
(17) $Timeout_p[r] \leftarrow \kappa + incarnation_p$
(18) reset $Timer_p(r)$ to $Timeout_p[r]$
(19) **else**
(20) replace in $Membership_p$ (r,v) by $(r, \max\{v,v'\}) : (r,v') \in Membership_q$
(21) **end if**
(22) reset $Timer_p(q)$ to $Timeout_p[q]$
(23) $update\_leader()$
(24) **if** $[leader_p = p]$ **then**
(25) collect $data_q$
(26) **end if**

Task 3:
(27) **upon expiration of** $Timer_p(q)$ **do**
(28) replace in $Membership_p$ (q,v) by $(q, v+1)$
(29) $Timeout_p[q] \leftarrow Timeout_p[q] + 1$
(30) reset $Timer_p(q)$ to $Timeout_p[q]$
(31) $update\_leader()$

Figure 8.5: Intra-region aggregator election and data aggregation (Alg. III).

Figure 8.5 presents the pseudocode executed by each sensor when it is up. The algorithm is the collection of n instances of this pseudocode, one for each sensor in the system. In Task 1, sensors broadcast messages periodically to try to become the aggregator, as well as to send their sensed data, with a periodicity of κ. Every message sent by a sensor p contains the set $Membership_p$ and $data_p$. In Task 2, if a sensor p receives a message (I-AM-ALIVE, q, $Membership_q$, $data_q$) with $q \neq p$ for the first time, it re-broadcasts the message to attempt reaching all the sensors of the region, updates $Membership_p$ based on $Membership_q$ (Lines 13-21), and resets $Timer_p(q)$. Then, p calls the procedure $update\_leader()$. Finally, if p is the aggregator, it collects $data_q$.

In Task 3, if $Timer_p(q)$ expires before a new I-AM-ALIVE message from q is received, then p "suspects" q. It replaces in $Membership_p$ (q, v) by $(q, v+1)$, increments $Timeout_p[q]$, resets the timer and calls $update\_leader()$. Observe that, if q has not crashed, upon reception of the next message from p, q will increment its associated counter in $Membership_q$.

The number of messages sent during a data acquisition period (κ) is quadratic in the number of sensors in the region $O(n^2)$, since every sensor broadcasts one message by Task 1, and sensors re-broadcast received messages.

Correctness Proof

Regarding the correctness proof of this algorithm, observe that it is very similar to the one in the previous section. The main differences are (1) the unknown membership and, (2) contrary to the previous algorithm, now we only require the existence of a multi-hop bidirectional path from some correct sensor(s) to the rest of the sensors. The first question (1) is addressed with a non-decreasing membership ($Membership_p$) and dynamically created timers, while the second (2) is overcome by re-broadcasting every message that a sensor receives for the first time (Line 12).

8.5 Global (Inter-Region) Level

As described in Figure 8.6, all the aggregators cannot usually communicate directly among them in order to collect all the data sensed in the different regions of the wide-area WSN. However, we assume that every pair of aggregators can communicate, either directly or indirectly (by re-broadcast). Based on this assumption, we implement an aggregator election and data aggregation algorithm for the global (inter-region) level. Figures 8.7 and 8.8 present the pseudocode executed by each sensor when it is up. The algorithm is the collection of n instances of this pseudocode, one for each sensor in the system.

The algorithm is an adaptation of the first algorithm for the local level, but executed only among the aggregators of the different regions to select the super-aggregator and to collect data of the whole sensor network. As in the first algorithm for the local level, we assume that aggregators do not need to know the identifiers of the rest of the aggregators in advance. With this algorithm, all the aggregators will select as super-aggregator the aggregator with the minimum incarnation number, using the aggregator identifier to break ties. Interestingly, the algorithm allows different regions to execute any of the three algorithms for the local (intra-region) level.

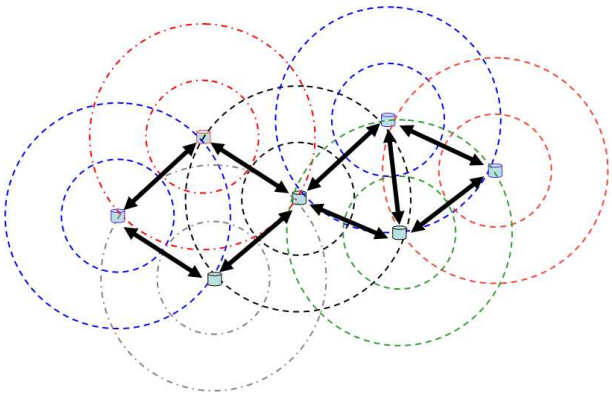

Figure 8.6: Large WSN divided in regions (only aggregators are shown).

The algorithm uses the variables $super\_leader_p$, $incarnation_{super\_leader}$ and also

Every sensor p executes the following:

Initialization (for both intra- and inter-region):
(1) **if** [intra-region algorithm is I] **then**
(2) add the following instruction to $GoToHibernation()$ (after Line 1):
(3) write ($super\_leader_p$, $incarnation_{super\_leader}$, $Timeout\_Super_p$) to stable storage
(4) execute the Initialization of Algorithm I (Lines 4-16 in Figure 8.2):
(5) read ($super\_leader_p$, $incarnation_{super\_leader}$, $Timeout\_Super_p$) from stable storage
(6) **else**
(7) **if** [intra-region algorithm is II] **then**
(8) execute the Initialization of Algorithm II (Lines 2-9 in Figure 8.4):
(9) **else if** [intra-region algorithm is III] **then**
(10) execute the Initialization of Algorithm III (Lines 2-6 in Figure 8.5):
(11) **end if**
(12) $super\_leader_p \leftarrow p$
(13) $incarnation_{super\_leader} \leftarrow incarnation_p$
(14) $Timeout\_Super_p \leftarrow \Delta_{OPERATION} + incarnation_p$
(15) **end if**
(16) **start tasks** 4, 5, 6 and 7

Task 4:
(17) **loop forever**
(18) **if** [$leader_p = p$] **then**
(19) $received\_from\_super \leftarrow FALSE$
(20) **if** [$super\_leader_p = p$] **then**
(21) **if** [intra-region algorithm is I] **then**
(22) wait ε time units
(23) **end if**
(24) broadcast (I-AM-THE-SUPER-LEADER, p, $incarnation_p$)
(25) **else**
(26) reset $timer\_super_p$ to $Timeout\_Super_p$
(27) **end if**
(28) **end if**
(29) wait $\Delta_{OPERATION}$ time units

Figure 8.7: Inter-region algorithm (Part I).

Task 5:
(30) **upon reception of** message (I-AM-THE-SUPER-LEADER, q, $incarnation_q$)
with $q \neq p$ for the first time **do**
(31) **if** [$leader_p = p$] **then**
(32) **if** [$incarnation_q < incarnation_{super\_leader}$] or
[($incarnation_q = incarnation_{super\_leader}$) and ($q \leq super\_leader_p$)] **then**
(33) $super\_leader_p \leftarrow q$
(34) $incarnation_{super\_leader} \leftarrow incarnation_q$
(35) broadcast (I-AM-THE-SUPER-LEADER, q, $incarnation_q$)
(36) broadcast (DIGEST, p)
(37) $received\_from\_super \leftarrow TRUE$
(38) **end if**
(39) **end if**

Task 6:
(40) **upon reception of** message (DIGEST, q) with $q \neq p$ for the first time **do**
(41) **if** [$leader_p = p$] **then**
(42) **if** [$super\_leader_p = p$] **then**
(43) collect DIGEST
(44) **else**
(45) broadcast (DIGEST, q)
(46) **end if**
(47) **end if**

Task 7:
(48) **upon expiration of** $timer\_super_p$ **do**
(49) **if** [$leader_p = p$] **then**
(50) **if** [$received\_from\_super = FALSE$] **then**
(51) $super\_leader_p \leftarrow p$
(52) $incarnation_{super\_leader} \leftarrow incarnation_p$
(53) $Timeout\_Super_p \leftarrow Timeout\_Super_p + \Delta_{TIMEOUT}$
(54) **end if**
(55) **end if**

Figure 8.8: Inter-region algorithm (Part II).

$Timeout\_Super_p$. When executed on top of the intra-region Algorithm I (Alg. I), these variables are read from stable storage during the execution of the initialization, and written in stable storage upon the execution of the $GoToHibernation()$ procedure; otherwise, they are initialized to the values p, $incarnation_p$ and $\Delta_{OPERATION} + incarnation_p$, respectively.

The proposed hierarchical algorithm works under the following assumptions:

- Eventually, there is a timely path (possibly with multiple hops) from the super-aggregator to the rest of the aggregators, as well as from every aggregator to the super-aggregator.
- The time $\Delta_{OPERATION}$ is set to a value such that eventually (1) the message broadcast by the super-aggregator (I-AM-THE-SUPER-LEADER) reaches the rest of the aggregators, and (2) the DIGEST message broadcast by every aggregator reaches the super-aggregator.
- $\Delta_{DATA\_ACQUISITION} \geq \Delta_{OPERATION}$ if this algorithm is executed combined with the first local-level algorithm (Alg. I).

The cost of the algorithm, measured as the number of messages sent during an operation period, is quadratic in the reduced number of aggregators (agg) to agree on a unique super-aggregator $O(agg^2)$ messages. This compares favourably to the case in which the WSN is composed of a unique region, where the cost would be quadratic, $O(n^2)$ messages, in the total number of sensors (n).

Observe that with this algorithm only aggregators agree on the common super-aggregator. Regular sensors (either eventually up or unstable) would not agree on the common super-aggregator. By one hand, since they are not aggregators within their respective regions, they do not disturb the super-aggregator election. On the other hand, due to the hierarchical structure of our implementation, it is sufficient that just the aggregators agree on a common super-aggregator.

Correctness Proof

The correctness proof of this algorithm is similar to the one of the algorithm in Figure 8.2. The main difference is the absence of direct communication among all the sensors in the network, and the existence of a path between every pair of aggregators. Due to this, the messages received for the first time by every aggregator are re-broadcast in order to reach the super-aggregator.

8.6 Energy-Aware Aggregator Election and Data Aggregation

From the hierarchical approach we have followed, we consider two energy levels for broadcasting messages, which correspond to the energy levels of the intra-region and the inter-region messages respectively. Graphically speaking, the radius of the biggest circles in Figure 8.3 and the small circles in Figure 8.6 are the same. Assuming that the distance between adjacent aggregators is approximately twice the radius of a region, we have that the energy level of inter-region messages must be approximately four times that of intra-region messages. However, the number of inter-region messages is usually small with respect to the total number of messages.

In the algorithms presented so far, eventually the aggregator sensor remains as aggregator until the end of its battery. Taking into account that the battery of a sensor that acts as aggregator decreases faster than the battery of a regular sensor (see Figure 8.9), in order to prevent the full depletion of the aggregator's battery, a battery depletion threshold can be introduced. When the aggregator detects a battery level below the depletion threshold, it induces the system to select another aggregator. This way the aggregator preserves some energy to act as a regular sensor.

Figure 8.10 presents the proposed modification, consisting in a new task that pe-

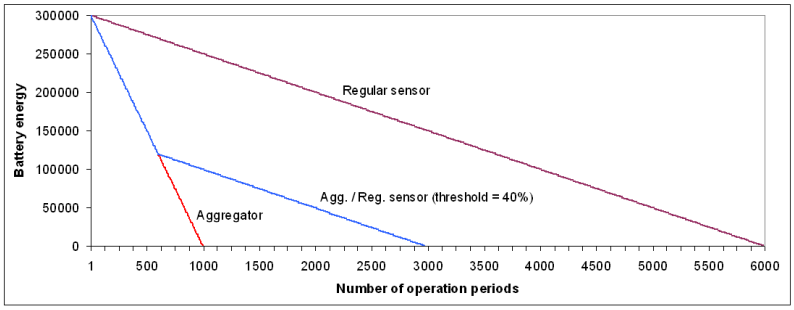

Figure 8.9: Battery life comparison for a sensor.

riodically checks if the sensor is the aggregator of its region and its battery level has dropped below the threshold. If it is the case, then the sensor increases either its incarnation number (in the case of the first local-level algorithm) or its suspicion counter (in the case of the second and third local-level algorithms).

Task 0:
(1) **loop forever**
(2) **if** $[leader_p = p]$ and $[getBatteryLevel() < THRESHOLD]$ **then**
(Alg. I) $incarnation_p \leftarrow incarnation_p + 1$
(Alg. II) $counter_p[p] \leftarrow counter_p[p] + 1$
(Alg. III) replace in $Membership_p$ (p, v) by $(p, v+1)$
(4) wait(γ)

Figure 8.10: Using a battery depletion threshold.

The selection of a certain threshold determines the QoS of the WSN, measured as the number of sensors that will remain active during a given period of time, provided the existence of a common aggregator. For a desired number of active sensors during a certain period, and according to sensors energy consumption, we can determine the associated threshold. The potential risk of this strategy occurs when, due to a battery level below the threshold, all the sensors start increasing their incarnation number or suspicion counter. When this happens, sensors could set their threshold to 0%, and the network can continue working properly, but without such QoS guarantee, until the end of the battery of all the sensors.

The use of a threshold provides an energy-aware aggregator election mechanism, since it allows a balanced depletion of the batteries of the (well-communicated) correct sensors of each region. As a consequence, the number of sensors that remain active during a given period of time increases, which is a relevant QoS measure in sensor networks.

Chapter 9

Conclusions and Future Work

Contents

9.1 Research Assessment

In this dissertation we have studied, for the first time, the Omega failure detector in the crash-recovery model. More specifically, we have focused on the design of algorithms that implement this failure detector in models of partial synchrony subject to crash-recovery failures. This research has led to four major contributions.

The redefinition of the Omega failure detector for the crash-recovery model. The definition of Omega is well suited to the crash model, but it can be improved in the crash-recovery model. The definition of Omega does not take into account unstable processes and hence they are allowed to permanently disagree with correct processes, which can be a serious drawback. For this reason we have defined the $Omega_{cr1}$ and $Omega_{cr2}$ failure detectors. Basically, the $Omega_{cr2}$ failure detector establishes that correct processes and unstable processes, when up, will agree on the same correct leader. With the $Omega_{cr1}$ failure detector, unstable processes do not trust any process upon recovery and if they trust a process it will be the correct leader. The $Omega_{cr2}$ failure detector requires a system where processes have access to stable storage while $Omega_{cr1}$ does not.

A collection of algorithms that implement *Omega*, $Omega_{cr1}$ or $Omega_{cr2}$. Our main contribution is a set of eight distributed algorithms that work in (slightly) different systems where processes are subject to crash-recovery failures. In this context, we have reflected on the limits of the synchrony required to implement $Omega_{cr1}$ and $Omega_{cr2}$. With regard to efficiency, we have implemented two communication-efficient algorithms: one for $Omega_{cr1}$ in a system without stable storage, based on nondecreasing local clocks; and another for $Omega_{cr2}$ where processes have access to stable storage.

Two algorithms implementing eventually perfect failure detectors. In the proposed distributed systems, subject to crash-recovery failures, it is not possible to implement a failure detector of the class $\diamond\mathcal{P}$. Basically, in such a system we cannot distinguish an unstable process from an eventually up (correct) process that has not yet

stabilized. For this reason, we have defined the $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$ failure detectors, which satisfy weaker properties but which are achievable in the crash-recovery model. In addition, we have presented two algorithms implementing $\diamond\mathcal{P}_{cr}$ and $\diamond\mathcal{P}_{k-cr}$. The algorithms rely strongly on the use of the leader election service provided by the $Omega_{cr2}$ failure detector.

Three aggregator election and data aggregation algorithms for wireless sensor networks. A wireless sensor network, WSN, can be seen as a distributed system subject to crash-recovery failures. On this basis, we have built three hierarchical aggregator election and data aggregation algorithms for large WSNs, on top of our implementations of the $Omega_{cr2}$ failure detector.

9.2 Future Work

The research carried out has led us to raise new questions on which we hope to work. We comment below on the direction of our research in the near future.

New Consensus algorithms. Properties satisfied by the $Omega_{cr1}$ and $Omega_{cr2}$ failure detectors can be used to implement Consensus algorithms in the crash-recovery failure model. It would be interesting to work on the design of efficient Consensus algorithms based on our implementations of the $Omega_{cr1}$ and $Omega_{cr2}$ failure detectors.

New Omega for different system models. Our interest in Omega leads us to study it in other distributed system models. With regard to the failure models, the most appealing are the Byzantine and the omission failure models.

The Byzantine failure model allows processes to behave arbitrarily. Processes can crash, crash and recover, deviate from the algorithm, act selfishly, lie and can even behave maliciously; i.e. as an adversary that tries to make the algorithm or protocol fail. This means that applications that are tolerant to Byzantine failures can be used in real systems that are open to the general public, such as the Internet.

The omission failure model is more restrictive than the crash-recovery model because, basically, it does not consider that a process can crash and recover, thus losing its status. There is increasing interest in this model because it is useful for studying security related problems. If we consider a system composed of untrusted processes which are equipped with trusted devices that allow the signing of messages, the failures that a malicious adversary can introduce would be limited to dropping signed messages. With this approach it would be possible to reduce some security problems usually studied in the Byzantine failure model, such as secure multi-party computation [126] and fair exchange [60], to canonical distributed problems, such as Consensus, in the omission model [39].

Nowadays there is an increasing demand for applications that allow collaboration and information sharing or acquiring in wide area systems. This involves a potentially huge number of distributed users and nodes, and, therefore, an underlying large-scale distributed system. In such a system, fault tolerance is essential and the study of Omega, focusing on scalability and performance, is of great interest.

Finally, we should not forget that computing devices are becoming more and more portable and that this portability must be supported by applications. This type of distributed system can be modelled as a system with *dynamic membership*; i.e. where processes can *join* and *leave* the system.

Application of failure detectors. As we have seen in this dissertation, failure detectors in general and Omega in particular can be used as a basis to solve problems other than Consensus. It would be interesting to study the applicability of $Omega_{cr1}$ and $Omega_{cr2}$ to existing agreement problems in the area of distributed systems, e.g. *k*-set agreement [33] and atomic commit [68, 120].

Bibliography

[1] The free on-line dictionary of computing. `http://foldoc.org/`.

[2] M. 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, 1997.

[3] M. Aguilera, W. Chen, and S. Toueg. Failure Detection and Consensus in the Crash-Recovery Model. *Distributed Computing*, 13(2):99–125, 2000.

[4] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S.Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, October 2008.

[5] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable Leader Election. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC'01)*, pages 108–122, Lisbon, Portugal, October 2001. LNCS 2180, Springer-Verlag.

[6] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 306–314, Boston, Massachusetts, July 2003.

[7] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 328–337, St. John's, Newfoundland, Canada, July 2004.

[8] M. Aguilera and S. Toueg. Failure Detection and Randomization: A Hybrid Approach to Solve Consensus. *SIAM J. Comput.*, 28(3):890–903, 1998.

[9] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, March 2002.

[10] J. Al-Karaki, R. Ul-Mustafa, and A. Kamal. Data aggregation in wireless sensor networks - Exact and approximate algorithms. In *Proceedings of IEEE Workshop on High Performance Switching and Routing (HPSR'04)*, pages 241–245, Phoenix, Arizona (USA), April 2004.

[11] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Johns Hopkins University, 1998.

[12] J. Aspnes. Fast deterministic consensus in a noisy environment. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, pages 299–308, 2000.

[13] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[14] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Symposium on Foundations of Computer Science (FODC'87)*, pages 337–346. IEEE Computer Society Press, October 1987.

[15] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.

[16] Ö. Babaoglu and S. Toueg. Non-Blocking Atomic Commitment. *Distributed Systems*, pages 147–166, 1993.

[17] 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, 1996.

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

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

[20] O. Biran, S. Moran, and S. Zaks. A Combinatorial Characterization of the Distributed Tasks Which Are Solvable in the Presence of One Faulty Processor. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 263–275, 1988.

[21] O. Biran, S. Moran, and S. Zaks. Tight Bounds on the Round Complexity of Distributed 1-Solvable Tasks. *Theor. Comput. Sci.*, 145(1-2):271–290, 1995.

[22] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[23] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[24] J. Burns, M. Gouda, and R. Miller. Stabilization and Pseudo-Stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[25] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[26] T. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[27] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, NY, USA, 1996. ACM.

[28] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[29] J. Chang and N. Maxemchuk. Reliable Broadcast Protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[30] B. Charron-Bost. Comparing the Atomic Commitment and Consensus Problems. *Future Directions in Distributed Computing*, pages 29–34, 2003.

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

[32] B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[33] S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.*, 105(1):132–158, 1993.

[34] W. Chen, S. Toueg, and M. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51:13–32, 2002.

[35] Y. Chen, A. Liestman, and J. Liu. Energy-Efficient Data Aggregation Hierarchy for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QSHINE'05)*, page 7, Lake Buena Vista, Florida (USA), August 2005. IEEE Computer Society Press.

[36] G. Chockler, S. Gilbert, and B. Patt-Shamir. Communication-Efficient Probabilistic Quorum Systems for Sensor Networks. In *Proceedings of the 4th IEEE Conference on Pervasive Computing and Communications Workshops (PerCom'06 Workshops)*, pages 111–117, Pisa, Italy, March 2006. IEEE Computer Society.

[37] B. Chor and C. Dwork. Randomization in Byzantine agreement, Randomness and Computation. *Advances in Computer Research*, 5:443–497, 1989.

[38] F. Chu. Reducing Omega to $\diamond\mathcal{W}$. *Information Processing Letters*, 67(6):289–293, September 1998.

[39] R. Cortiñas, F. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L. Draque, and I. Soraluze. Secure Failure Detection in TrustedPals. In *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, pages 173–188, 2007.

[40] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Inf. Comput.*, 118(1):158–179, 1995.

[41] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[42] C. Delporte-Gallet, S. Devismes, and H. Fauconnier. Robust Stabilizing Leader Election. In *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, pages 219–233, 2007.

[43] E. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[44] D. Dolev. The Byzantine Generals Strike Again. *J. Algorithms*, 3(1):14–30, 1982.

[45] 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.

[46] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure Detectors in Omission Failure Environments. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, page 286, Santa Barbara, California, USA, August 1997.

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

[48] P. Dutta and R. Guerraoui. Fast Indulgent Consensus with Zero Degradation. In *Proceedings of the 4th European Dependable Computing Conference (EDCC'02)*, pages 191–208. Springer-Verlag, 2002.

[49] P. Dutta, R. Guerraoui, and B. Pochon. Fast Non-Blocking Atomic Commit: An Inherent Trade-off. *Information Processing Letters*, 91(4):195–200, 2004.

[50] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[51] R. Ekwall, A. Schiper, and P. Urbán. Token-based Atomic Broadcast using Unreliable Failure Detectors. In *Proceedings of the 23th International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 52–65, 2004.

[52] P. Ezhilchelvan, D. Palmer, and M. Raynal. An Optimal Atomic Broadcast Protocol and an Implementation Framework. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, pages 32–39, 2003.

[53] P. Feldman and S. Micali. An Optimal Probabilistic Algorithm For Synchronous Byzantine Agreement. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, pages 341–378, 1989.

[54] A. Fernández, E. Jiménez, and S. Arévalo. Minimal System Conditions to Implement Unreliable Failure Detectors. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 63–72, 2006.

[55] A. Fernández, E. Jiménez, and M. Raynal. Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'06)*, pages 166–178, Philadelphia, Pennsylvania, June 2006.

[56] A. Fernández and M. Raynal. From an Intermittent Rotating Star to a Leader. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS'07)*, pages 189–203, Guadeloupe, French West Indies, December 2007. LNCS 4878, Springer-Verlag.

[57] M. Fischer. The consensus problem in unreliable distributed systems (a brief survey). *Foundations of Computation Theory*, 158:127–140, 1983.

[58] M. Fischer, N. Lynch, and M. Merritt. Easy Impossibility Proofs for Distributed Consensus Problems. *Distributed Computing*, 1(1):26–39, 1986.

[59] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[60] F. Freiling, M. Herlihy, and L. Draque. Optimal Randomized Fair Exchange with Secret Shared Coins. In *Proceedings of the 9th International Conference On Principles Of Distributed Systems (OPODIS'05)*, pages 61–72, 2005.

[61] F. Freiling, C. Lambertz, and M. Majster-Cederbaum. Modular Consensus Algorithms for the Crash-Recovery Model. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pages 287–292, 2009.

[62] E. Gafni. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 143–152, 1998.

[63] E. Gafni and L. Lamport. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*, pages 330–344, 2000.

[64] R. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California at Santa Cruz, Santa Cruz, CA, USA, 1992.

[65] 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, Le Mont-Saint-Michel, France, September 1995. LNCS 972, Springer-Verlag.

[66] R. Guerraoui, M. Hurfin, A. Mostéfaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in Asynchronous Distributed Systems: A Concise Guided Tour. *Advances in Distributed Systems*, pages 33–47, 1999.

[67] R. Guerraoui and P. Kouznetsov. On the Weakest Failure Detector for Non-Blocking Atomic Commit. In *Proceedings of the 2nd International IFIP Conference on Theoretical Computer Science (TCS'02)*, pages 461–473, 2002.

[68] R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proceedings of the 14th Symposium on Reliable Distributed Systems (SRDS'95)*, pages 41–51, Bad Neuenahr, Germany, September 1995.

[69] R. Guerraoui, R. Olivera, and A. Schiper. Stubborn Communication Channels. Technical Report CNDS–98–4, École Polytechnique Fédérale de Lausanne, Switzerland, 1996.

[70] R. Guerraoui and M. Raynal. The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers*, 53(4):453–466, April 2004.

[71] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. *Distributed Systems (2nd Ed.)*, pages 97–145, 1993. Expanded version appeared as a Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca, NY, 1994.

[72] J. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the 3th Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 50–61, 1984.

[73] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas, and R. Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the 11th ACM Annual ACM*

Symposium on Parallel Algorithms and Architectures (SPAA'99), pages 251–260, March 1999.

[74] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the Hawaiian International Conference on Systems Science (HICSS'00)*, pages 35–42, January 2000.

[75] F. Hu, X. Cao, and C. May. Optimized Scheduling for Data Aggregation in Wireless Sensor Networks. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 2, pages 557–561, Las Vegas, Nevada (USA), April 2005. IEEE Computer Society.

[76] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in Asynchronous Systems Where Processes Can Crash and Recover. In *Symposium on Reliable Distributed Systems (SRDS'98)*, pages 280–286, West Lafayette, Indiana, USA, October 1998.

[77] 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.

[78] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Chasing the Weakest System Model for Implementing Omega and Consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269–281, 2009.

[79] E. Jiménez, S. Arévalo, and A. Fernández. Implementing the Ω Failure Detector with Unknown Membership and Weak Synchrony. Technical Report RoSaC–2005–2, Universidad Rey Juan Carlos, Spain, January 2005.

[80] E. Jiménez, S. Arévalo, and A. Fernández. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60–63, 2006.

[81] I. Keidar and S. Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial. In *Proceedings of the 1st Latin-American Symposium in Dependable Computing (LADC'03)*, pages 366–368, 2003.

[82] B. Krishnamachari, D. Estrin, and S. Wicker. The Impact of Data Aggregation in Wireless Sensor Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems, Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 575–578, Vienna, Austria, July 2002. IEEE Computer Society.

[83] M. Kumar, L. Schwiebert, and M. Brockmeyer. Efficient Data Aggregation Middleware for Wireless Sensor Networks. In *Proceedings of the First International Conference on Mobile, Ad-Hoc, and Sensor Systems (MASS'04)*, pages 579–581, Ft. Lauderdale, Florida (USA), October 2004. IEEE Computer Society Press.

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

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

[86] L. Lamport. Lower bounds for asynchronous consensus. Technical Report MSRTR-2004-72, Microsoft Research, 2004.

[87] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[88] M. Larrea, A. Fernández, and S. Arévalo. Optimal Implementation of the Weakest Failure Detector for Solving Consensus. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 52–59, Nurnberg , Germany, 2000.

[89] M. Larrea, A. Fernández, and S. Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 53(7):815–828, July 2004.

[90] M. Larrea, A. Fernández, and S. Arévalo. Eventually Consistent Failure Detectors. *Journal of Parallel and Distributed Computing*, 65(3):361–373, March 2005.

[91] M. Larrea and A. Lafuente. Brief announcement: Communication-efficient implementation of failure detector classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, pages 495–496, Krakow, Poland, September 2005. LNCS 3724, Springer-Verlag.

[92] S. Lindsey and C. Raghavendra. PEGASIS: Power-efficient gathering in sensor information systems. In *Proceedings of the IEEE Aerospace Conference*, pages 1125–1130, March 2002.

[93] S. Lindsey, C. Raghavendra, and K. Sivalingam. Data Gathering Algorithms in Sensor Networks Using Energy Metrics. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):924–935, 2002.

[94] N. Lynch. Distributed Algorithms. *Morgan Kaufmann*, 1996.

[95] D. Malkhi, F. Oprea, and L. Zhou. Omega Meets Paxos: Leader Election and Stability Without Eventual Timely Links. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, pages 199–213, Krakow, Poland, September 2005. LNCS 3724, Springer-Verlag.

[96] D. Malkhi, F. Oprea, and L. Zhou. Omega Meets Paxos: Leader Election and Stability Without Eventual Timely Links. Technical Report MSR-TR-2005-93, Microsoft Research, 2005.

[97] 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.

[98] N. Malpani, J. Welch, and N. Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Proceedings of the 4th International Workshop on Algorithms and Methods for Mobile Computing and Communications*, pages 96–103, August 2000.

[99] S. Masum, A. Ali, and M. Bhuiyan. Asynchronous Leader Election in Mobile Ad Hoc Networks. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 2 (AINA'06)*, pages 827–831, 2006.

[100] L. Moser, P. Melliar-Smith, and V. Agrawala. Processor Membership in Asynchronous Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 5(5):459–473, 1994.

[101] A. Mostéfaoui, E. Mourgaya, and M. Raynal. Asynchronous Implementation of Failure Detectors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'03)*, pages 351–360, San Francisco, California, June 2003.

[102] A. Mostéfaoui, E. Mourgaya, M. Raynal, and C. Travers. A Time-free Assumption to Implement Eventual Leadership. *Parallel Processing Letters*, 16(2):189–208, 2006.

[103] A. Mostéfaoui and M. Raynal. Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[104] A. Mostéfaoui, M. Raynal, and C. Travers. Time-Free and Timer-Based Assumptions Can Be Combined to Obtain Eventual Leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656–666, July 2006.

[105] J. Neander, E. Hansen, M. Nolin, and M. Björkman. Asymmetric Multihop Communication in Large Sensor Networks. In *Proceedings of the International Symposium on Wireless Pervasive Computing (ISWPC'06)*, Phuket, Thailand, January 2006.

[106] R. Oliveira. *Solving consensus: from fair-lossy channels to crash-recovery of processes*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000.

[107] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, Swiss Federal Institute of Technology, Lausanne, 1997.

[108] V. Park and M. Corson. A Highly Adaptative Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of the 16th IEEE International Conference on Computer Communications (INFOCOM'97)*, pages 1405–1413, April 1997.

[109] S. Patil and S. Das. Serial data aggregation using space-filling curves in wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 326–327, Los Angeles, California (USA), November 2003. ACM Press.

[110] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[111] S. Pleisch, O. Rütti, and A. Schiper. On the Specification of Partitionable Group Membership. In *Proceedings of the 7th European Dependable Computing Conference (EDCC'08)*, pages 37–45, 2008.

[112] B. Przydatek, D. Xiaodong Song, and A. Perrig. SIA: secure information aggregation in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 255–265, Los Angeles, California (USA), November 2003. ACM.

[113] M. Rabin. Randomized Byzantine Generals. In *Proceedings of the 24th Symposium on Foundations of Computer Science (FOCS'83)*, pages 403–409, 1983.

[114] M. Raynal. Eventual Leader Service in Unreliable Asynchronous Systems: Why? How? In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA'07)*, pages 11–24, 2007.

[115] L. Sabel and K. Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical Report TR95-1488, Cornell University, Ithaca, 1995.

[116] N. Santoro and P. Widmayer. Time is Not a Healer. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science (STACS'89)*, pages 304–313, 1989.

[117] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(3):149–157, 1997.

[118] A. Schiper and S. Toueg. From Set Membership to Group Membership: A Separation of Concerns. *IEEE Trans. Dependable Sec. Comput.*, 3(1):2–12, 2006.

[119] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[120] D. Skeen. Nonblocking Commit Protocols. In *Proceedings of the International Conference on Management of Data (SIGMOD'81)*, pages 133–142, 1981.

[121] S. Toueg. Randomized Byzantine Agreements. In *Proceedings of the 3th Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 163–178, 1984.

[122] P. Tsuchiya. The Landmark Hierarchy: A new hierarchy for routing in very large networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'88)*, pages 35–42, 1988.

[123] K. Vaidyanathan, S. Sur, S. Narravula, and P. Sinha. Data aggregation techniques in sensor networks. Technical Report 11/04-TR60, The Ohio State University, November 2004.

[124] S. Vasudevan, J. Kurose, and D. Towsley. Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks. In *Proceedings of the 12th International Conference on Network Protocols (ICNP'04)*, pages 350–360, October 2004.

[125] J. Widder and U. Schmid. The Theta-Model: achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

[126] A. Yao. Protocols for Secure Computations (Extended Abstract). In *Proceedings of the 23th Symposium on Foundations of Computer Science (FOCS'82)*, pages 160–164, 1982.

[127] 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)*, pages 201–208, 1996.