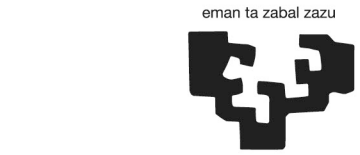


Failure Detectors and Communication Efficiency in the Crash and General Omission Failure Models

Dissertation for the degree of
Doctor of Philosophy presented by

Roberto Cortiñas

advisors

Alberto Lafuente - Iratxe Soraluze

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

January, 2011

ISBN: 978-84-694-5838-9

Abstract

Consensus is one of the fundamental problems in fault tolerant distributed systems. In addition to the importance of the problem itself, consensus can be a way to solve many other problems in distributed systems, so it is considered a key topic in the Distributed Computing area.

Although many solutions have been proposed to solve consensus in synchronous systems, *[Fischer, Lynch, and Paterson, 1985]* presented an impossibility result, namely Fischer-Lynch-Paterson or FLP, that states that it is impossible to reach consensus in asynchronous systems where even one process may crash. In order to circumvent FLP, *[Chandra and Toueg, 1996]* proposed the *unreliable failure detector* abstraction, which has been widely studied in several systems, specially those where processes can only fail by crashing.

Failure detectors offer a modular approach that allows other applications such as consensus to use them as a building block. Additionally, the failure detector abstraction allows to encapsulate the synchrony assumptions of the system, so that applications which make use of failure detectors can be designed as if they run in pure asynchronous systems.

In this work we show that failure detectors can also be applied to the general omission failure model, in which processes may fail by crashing and by omitting messages either when sending or receiving. As a practical example, we propose a solution to a security area problem called *Secure Multiparty Computation* by using failure detectors for general omission.

In the context of failure detectors in the crash failure model we also study *communication efficiency*, a performance measure achieved when there are only n links that carry messages forever, being n the number of processes. We improve this measure by defining *communication optimality*, in which only c links are needed, being c the number of correct processes. In this regard, we propose some communication-optimal implementations of the eventually perfect failure detector class $\diamond\mathcal{P}$. Finally, we propose a communication-efficient implementation of a failure detector for the general omission failure model. In this case, we define communication efficiency as a linear number of links carrying messages forever.

Acknowledgements

This dissertation would not have been possible to complete without the help of so many people. I would like to say *thank you* to every one who helped me and those who would have liked to help me but I did not give the chance of doing so. Nevertheless, it would take too long to thank it properly, so I will try to sum it up. I apologize beforehand if omit someone.

I must begin thanking my advisors, Alberto and Iratxe. Starting from Alberto, for his generous support from the very beginning and his perseverance on helping me, in spite of the hard moments. In the same sense, I would also like to thank Iratxe for her help and assistance whenever I asked her. Many thanks also to Mikel for his *countless* and unconditional help.

I am also very grateful to Felix Freiling, for his support during my stay at Mannheim and, specially, for his advise since we met first.

I should not forget my workmates, who have added some very valuable warmth to an otherwise solitary and hard work.

Of course, I would also like to thank my family, although it would take too long to explain all the reasons, which go far beyond this thesis.

Finally, Olatz; thanks for all the previous reasons and far much more. You have suffered a lot of the bad moments and inconveniences of this thesis, but you have always kept supporting and encouraging me. Thanks to Ibai for teaching me taking more advantage of my time (parents know what I am talking about) and for helping me to realize what is really important in life.

Preface

The work presented in this dissertation is the result of the research carried out in the Distributed Systems Group, at the University of the Basque Country (UPV/EHU). Additionally, some parts related to the general omission failure model were also developed in collaboration with researchers from the University of Mannheim.

Some of the researching results of this work have been published in several conferences and journals. In this preface we present a brief abstract of each publication resulting from this work, classified by the chapter(s) they are related to:

- Communication efficiency in the crash failure model (Chapter 5).
  - *[Larrea, Lafuente, Soraluze, Cortiñas, and Wieland, 2007a]* introduces *communication optimality* and includes some implementations of the $\diamond\mathcal{P}$ failure detector class.
  - *[Larrea, Soraluze, Cortiñas, and Lafuente, 2008]* includes an evaluation of communication-optimal $\diamond\mathcal{P}$ implementations presented in *[Larrea et al., 2007a]*.
  - *[Lafuente, Larrea, Soraluze, and Cortiñas, 2008]* focuses on communication locality and low sporadic overhead of communication-optimal $\diamond\mathcal{P}$ algorithms.

  The main contents of these works were wrapped up into a paper submitted to the Elsevier *Journal of Computer and System Sciences* (JCSS) in 2009.

- Failure detectors for the general omission failure model (Chapters 6 and 7).
  - *[Cortiñas, Freiling, Ghajar-Azadanlou, Lafuente, Larrea, Penso, and Soraluze, 2007]* addresses *Secure Multiparty Computation* (SMC) in the Byzantine failure model, problem which is turned into solving consensus in the general omission failure model by using a secure platform called *TrustedPals*. A new failure detector class for the general omission failure model is presented, together with an implementation of a failure detector of this class. We present a redefinition of the consensus properties for the general omission failure model and propose an adaptation of the consensus algorithm from *[Chandra and Toueg, 1996]* to work with the proposed failure detector class.

  An extended work from *[Cortiñas et al., 2007]* was submitted to the IEEE *Transactions on Dependable and Secure Computing* journal (TDSC) in 2010.

- Communication efficiency in the general omission failure model (Chapter 8).
  - *[Cortiñas, Soraluze, Lafuente, and Larrea, 2010]* proposes a communication-efficient implementation of the eventually perfect failure detector class $\diamond\mathcal{P}$ in the general omission failure model.

  An extended version of this work was submitted to the Elsevier *Information Processing Letters* (IPL) journal in 2010 (update: the paper, *[Soraluze, Cortiñas, Lafuente, Larrea, and Freiling, 2011]*, was accepted in December, 2010).

Table 1 summarizes the list of publications cited before.

Publication	Conference/Journal	Core/Rank	Chapters
[Larrea et al., 2007a]	LADC 2007	-	5
[Larrea et al., 2008]	PDP 2008	C	5
[Lafuente et al., 2008]	PODC 2008	A	5
[Cortiñas et al., 2007]	SSS 2007	C	6,7
[Cortiñas et al., 2010]	PODC 2010	A	8
[Soraluze et al., 2011]	Elsevier IPL 2011 (in press)	B	8

Table 1: List of publications resulting from this work

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

NOWADAYS computing systems are present in almost every task to accomplish. However, computers and networks that form systems are prone to faults, so the service they provide can be interrupted. Some systems can afford temporary breaks of service, but many others require a higher degree of *dependability*.

Dependability was defined by *[Laprie, Avizienis, and Kopetz, 1992]* as trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers. According to *[Laprie et al., 1992; Avizienis, Laprie, Randell, and Landwehr, 2004]*, dependability has several attributes, which are differently emphasized depending on the considered application. In our case and following *[Jalote, 1994]*, the most relevant attributes of dependability are *reliability*, *availability*, *safety* and *security*. Reliability implies *continuity of service*, whereas availability deals with *readiness for usage* and safety with *avoidance of catastrophic consequences on the environment*. Security is a composite of several attributes, since it asks for confidentiality and availability for (only) authorized actions, in addition to the non-occurrence of improper (non-authorized) alterations of information, i.e., *integrity*.

[Avizienis et al., 2004] proposed an alternate definition of dependability as the ability to avoid service failures that are more frequent and more severe than is acceptable. In this regard, reliability is the most significant attribute when considering the resilience of a system against failures, i.e., *fault tolerance*. Fault tolerance is defined as the ability of a system to provide service as expected (or with a minimal degradation) even if some components fail. In this sense, fault tolerance allows to build dependable systems from undependable components. *[Gärtner, 1999]* emphasized that to be able to tolerate faults, it is necessary a form of *redundancy*. If a vital component of the system fails and there is no possible replacement (redundancy), then the whole service could be compromised. For example, server replication or checksums are different forms of redundancy. Hence, the more redundancy we add to the system, the more fault-tolerant the system can be. However, redundancy involves a cost, so it should be used carefully.

Additionally, *[Gärtner, 1999]* stated that redundancy is necessary for fault tolerant, but not sufficient, and emphasized the need of detection and correction of faults.

In this regard, distributed systems usually present a better fault tolerance than other alternatives such as centralized systems. Informally, a *distributed system* consists of a set of processes which collaborate to achieve a common goal by communicating through message passing.[1]

Coordination in distributed systems is a fundamental issue, since every component in the system has only a partial view of the global state. More precisely, processes in a distributed system usually have to agree on some value or task. There are many agreement problems with different properties, however, the problem of *consensus* is considered the paradigm of agreement. In the consensus problem, participating processes first propose a value and then every (correct) participating process has to agree (decide) on one of the previously proposed values. Although the problem seems simple, *[Fischer et al., 1985]* proved that consensus has no deterministic solution in environments with crash type failures and in which timing assumptions can not be made (also called *asynchronous systems*). In order to circumvent that impossibility result, *[Chandra and Toueg, 1991]* proposed the *failure detector* abstraction.

Failure detectors have been widely studied since they were first proposed in 1991. As an example of its relevance, we could point out that the authors have recently been awarded with the 2010 *Edsger W. Dijkstra* Prize in Distributed Computing (*[Aguilera and Raynal, 2010]*).

In this work we explore the application of failure detectors in different scenarios. First, we study a communication measure regarding sent messages, namely *communication efficiency*, in the traditional failure model of failure detectors, the crash failure model. Then we show how failure detectors can also be implemented in another failure model; the general omission. Following this path, we prove that failure detectors in the general omission model can be useful to solve a security problem called *Secure Multiparty Computation* (SMC). Finally, we study communication efficiency for the general omission failure model as well.

1.2 Contributions of this Work

In this work we make some contributions to the failure detector area:

- First, we improve the implementation of failure detectors in the crash failure model according to their communication. In this regard, starting from the concept of *communication efficiency* previously presented by other authors, we define *communication optimality*, which improves the previous one, and present some implementations of the eventually perfect failure detector class $\diamond\mathcal{P}$ adapted to different scenarios.

[1] Some authors consider other communicating mechanisms, such as shared memory access, but in this work we focus on message passing.

- We implement failure detectors in a weaker failure model; the general omission failure model. First we present some issues regarding failure detection in the general omission failure model in partially synchronous systems. Then we define a failure detector class adapted to this model and present an implementation of a failure detector belonging to that class. We also show how a consensus algorithm devised to work with a failure detector in the crash failure model can be adapted to work with the previously defined failure detector for the general omission failure model.
- In order to show a case of use of the previously implemented failure detector in the general omission model, we present an example in which a Byzantine model can be reduced to a general omission model. This way, the failure detector of the previous contribution can be directly applied to provide a modular solution.
- Finally, we also look for communication efficiency in the general omission model and present a communication-efficient implementation of a failure detector in that failure model.

1.3 Organization of the Document

This document is structured as follows:

Chapter 1. In the first chapter we offer some hints about general concepts of distributed systems and fault tolerance.

Chapters 2, 3 and 4. These chapters introduce a background of the topics referenced in the dissertation. Chapter 2 presents concepts related to the definition of system models. More precisely, we mainly focus on processes, channels, timing models and process failure models. Chapters 3 and 4 introduce the consensus problem and the failure detector abstraction respectively.

Chapter 5. In this chapter we address communication efficiency when implementing failure detectors in the crash failure model. We define *communication optimality* and show how it can be achieved through some implementations.

Chapter 6. This chapter focuses on the application of failure detectors to the general omission failure model. We first introduce the general omission failure model and present some issues related to failure detection in systems with such a failure model. We then propose a new class of failure detectors for the general omission failure model, together with a general implementation. Additionally, we show how to adapt the consensus algorithm from *[Chandra and Toueg, 1996]* in order to work with the defined failure detector class.

Chapter 7. In this chapter we show that the failure detector for the general omission failure model presented in the previous chapter can be used as a building block to provide the consensus problem with a solution in the Byzantine failure model. More precisely, we initially present a security problem called Secure Multiparty Computation (SMC). Then, we show that SMC in the asynchronous model with Byzantine failures can be reduced to consensus in the general omission failure model by using a platform called TrustedPals. In fact, the achieved system matches the system model of Chapter 6, so we propose to use the failure detector proposed in that chapter as a building block to solve the problem of this chapter.

Chapter 8. This chapter presents a communication-efficient implementation of a failure detector in the general omission failure model. In this work communication efficiency is defined in terms of a linear number of links carrying messages forever. As in Chapter 6, we show that the consensus algorithm from *[Chandra and Toueg, 1996]* can also be adapted to work with this communication-efficient failure detector.

Chapter 9. This chapter wraps up the results presented in this work and suggests directions for future work.

Chapter 2

System Models

ROUGHLY speaking, a system model is a collection of assumptions that allow us to define the characteristics, assumptions and constraints a system might exhibit, such as timeliness or failure patterns. When devising an algorithm, one of the most important tasks is to define the system model in which that algorithm should work. If later we want to implement that algorithm in a real system, it will be sufficient to know if that particular real system meets the characteristics of the system model the algorithm was devised for. In this regard, the system should be realistic enough to support real life applications but should also lead to implementable systems. In this chapter we will present the main abstractions that allow to define the system models we will work on.

Outline. First, Section 2.1 presents a brief introduction system models. In Section 2.2 the *process* abstraction is addressed, whereas the *communication channel* abstraction is presented is Section 2.3. Time related assumptions are introduced in Section 2.4. Finally, main process failure models are described in Section 2.5.

2.1 Introduction

Models are fundamental when designing and evaluating solutions to a given problem. Informally, a model is an abstraction of an object of interest. A good model should point out the relevant aspects of the object, leaving out the irrelevant ones. *[Schneider, 1993]* stated that a good model should be *accurate* and *tractable*. A model is accurate if evaluations based on the model are actually valid for the object (and not only for the model). Tractable implies that evaluations are possible.

Components of a distributed system can be modelled by describing their possible behaviour. Before modelling such behaviour, we should define the following concepts:

- State. The state of a distributed system is composed of states of individual components (i.e., processes and channels).

- Trace. The behaviour of a system is represented by an infinite sequence of system states. A trace represents a possible execution of a system.
- Step. A pair of successive states, i.e., possible transitions between states, is called a step.

The behaviour of a system is represented by *traces*, i.e., sequences of *states*. A *property*[1] is a set of such traces and represents a component or the distributed system as a whole.

Defining properties is a very useful technique, since it makes an abstraction independent from its possible implementations. Hence, any implementation that meets the necessary properties satisfies the requirements. In the same way, the solution can be designed independently from any particular system, e.g., delays in communication or speed of processes.

When defining properties, it is usual to look for the weakest solution in a given system in order to know what the minimal properties should be met for a set of given assumptions.

[Lamport, 1977] proposed a classification of properties into two classes: *safety* and *liveness*. Identifying the properties of a system into these classes leads to a better understanding of those properties, better specifications and, consequently, clear proofs of possible implementations. Useful distributed systems should provide both types of properties:

- A *liveness* property aims that some particular *good* thing will eventually happen (for instance, the program will eventually produce a result). In our case, a property of this type will *eventually* be met.
- A *safety* property states that some particular *bad* thing never happens (for example, the program will never produce a wrong result). In our case, that implies that a property of this type is continuously met (in all prefixes of a given execution).

[Alpern and Schneider, 1985] gave formal definitions of safety and liveness and stated that properties of dependable systems can be classified into one or both classes. In this sense, note that a property could be classified as a combination of both liveness and safety, i.e., in some cases a property can be classified into both classes.

Further detailed information about safety and liveness can be found in *[Lamport, 1977 and 1979; Alpern and Schneider, 1985 and 1987; Charron-Bost, Toueg, and Basu, 2000; Benenson, Freiling, Holz, Kesdogan, and Penso, 2006b]*.

Uniformity. There is an interesting property, called *uniformity*, which usually is applied to a given property and which states that the property is hold by every process in the system (regardless of being correct or faulty). Uniformity was introduced by *[Neiger and Toueg, 1993]* and used at length by *[Hadzilacos and Toueg, 1994]* in the context of broadcast primitives. Observe that uniform properties are quite interesting since they consider every process as being potentially correct as long as it does not fail.

[1] Some authors consider conditions instead

2.2 Processes

A process can be defined as a computational entity. Processes execute user-level applications, which make use of the distributed algorithms we will focus on. Thus every process executes a copy of the same distributed algorithm and keeps a local state of that execution without loss of generality.

Sometimes a *process* is also called *processor* or *site*. It is normally assumed that there is only one process per processor or site.

In this work we will model a distributed system as a set Π of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, which are connected through pairwise bidirectional communication channels in a fully connected topology. Sometimes we will also use p, q, r, etc. to denote processes. n denotes the number of processes in the system. Note that $n = |\Pi|$.

The set of processes is known, finite and does not change during the execution of the system.

We consider processes as *correct* or *incorrect* depending on their characteristics related to the failure model of the system (see Section 2.5):

- *Correct* process: process that behaves according to its specification, i.e., process that does not experience any failure. We denote the number of correct processes in the system by c.
- *Incorrect* process, namely *faulty*: process that suffers a failure that might affect its specification, e.g., a crashed process will not be able to compute any more. In Section 2.5 we will present some classical failure models according to the types of failures that may occur in the system. We denote the number of faulty processes in the system by f.

Clearly $n = c + f$. It is usual to assume that there is always at least one correct process in (every execution of) the system.

2.2.1 Algorithms, Events and Runs

As stated before, processes execute algorithms. An algorithm A consists of a set of deterministic automata, one per each process. As a general rule, in the rest of the chapters of this work we will give our algorithms in an event-based notation and thus assume that a local FIFO event queue is part of the local state of every process. Within an execution step, a process takes an event from the queue, performs a state transition according to the event, and then may send a message or add a new event to the queue. Message sends and arrivals are treated as events too, i.e., when a message arrives, an appropriate event is added to the queue. We say that the message is received[2] by the process when this event is processed. We assume here that every process which does not crash executes infinitely many events.

[2]We will distinguish *deliver* and *receive* in the next section

A run of an algorithm is the combination of followed steps and state information of a given execution.

Event generation. There are two classical approaches for event generation; time-driven and message-driven algorithms. In *time-driven* algorithms events can be triggered by using the passage of time (measured by clocks or/and timers). On the other hand, in *message-driven* algorithms event generation only relies on message reception. In this work we will follow time-driven approaches.

2.3 Communication Channels/Links

Informally, communication channels or links represent an abstraction of the network. Processes communicate among them by sending messages though their corresponding links. *Channel* and *link* concepts may often be used interchangeably, but in some cases *channel* implies a higher level of abstraction, as presented in Chapter 7.

An unidirectional communication link from a process p to another process q allows p to send messages to q. When a link from p to q is bidirectional, it is assumed a communication link from p to q and another one from q to p. We denote the link from p to q by c_{pq}. In this dissertation we will assume a bidirectional communication channel/link between every pair of processes, that is, for every p and q, there will be a link c_{pq}.

Figure 2.1 presents a classical architecture of a distributed system; processes which communicate through a network (communication channels).

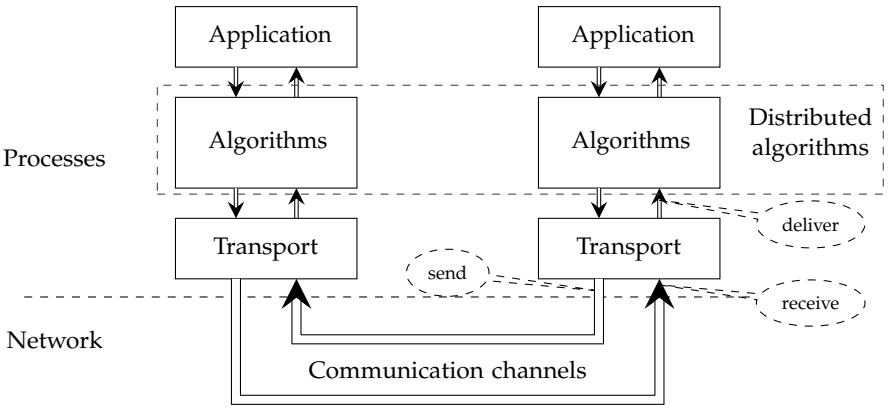


Figure 2.1: Layers in a distributed system

Observe that we distinguish *receive* and *deliver*. *Receiving* implies the arrival of a message at a process. *Delivering* is the previous step before processing a message. As we will see in Section 2.5.3 messages can be omitted after *receiving* and before *delivering*.

Every sent message is labelled with a unique identifier.

Links can be assumed to fulfil different properties, so they can be classified into different types. We will analyse *reliable links* and *fair lossy links*.

2.3.1 Reliable Links

Also called *perfect links*. With this type of link every sent message to a correct process is received, i.e., there is no message loss. Additionally, no message is created, corrupted or duplicated by the channel. Formally, reliable links offer the following properties (*[Basu, Charron-Bost, and Toueg, 1996]*):

- No creation. Every received message has been previously sent.
- No duplication. No message is received by a process more than once.
- No loss. If a process p sends a message m to another process q and q is correct[3] then q will eventually receive m.

Observe that the *no creation* and *no duplication* properties are safety properties, whereas *no loss* is a liveness properties.

Since the implementation of reliable links is not trivial, *eventual reliable* links, also known as *quasi-reliable* or *correct-restricted* channels, have also been proposed. An eventual reliable channel is a reliable link which corresponding processes are correct.

2.3.2 Fair Lossy Links

Also called *fair-loss* links. Informally, if a infinite number of messages is sent through a fair-lossy link, then it may drop an infinite number of messages, but an infinite subset of messages will be received. As a consequence, if a message is sent infinitely often, then the receiver will eventually receive it.

Formally, a fair lossy link must meet the following properties:

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

Sometimes, the *uniform integrity* property is used instead of the *no creation* and *finite duplication* properties:

- Uniform integrity. If q receives a message m from p then p previously sent m to q; and if q receives m infinitely often from p, then p sends m infinitely often to q.

[Basu et al., 1996] showed that in the crash failure model reliable links can be simulated using fair lossy links and a majority of correct processes. Hence, any problem solvable by using reliable links is also solvable by using fair lossy links.

[3] So q will execute *receive* actions infinitely often (original definition)

Eventually reliable links. An eventually reliable link can lose messages (fair lossy link), but there is a time after which all sent messages are eventually received (reliable link). Observe that an eventually reliable link can lose an unbounded but finite number of messages. This link presents the next properties:

- No creation.
- No duplication.
- Finite loss. If q is correct, the number of messages sent by p to q that are not received by q is finite.

[Basu et al., 1996] proved that reliable links are strictly stronger than eventually reliable links, i.e., problems solvable with reliable links may not be solved with eventually reliable links.

2.4 Time and Timing Models

In this section we address one of the most important subjects when defining a system model; its *timing assumptions*. This subject covers many of the characteristics of the system, such as communication delays or relative process speeds.

First, we will introduce the *clock* concept and then we will present the *synchronous*, *asynchronous* and *partially synchronous* timing models.

Clocks

Every process is supposed to have a local clock that allows to measure time passage. However, in a distributed system it is not always possible to keep all the clocks synchronized, due to the fact that, for example, every process clock could have its own drift or deviation from a perfect clock and in asynchronous systems such parameters can be unbounded. *[Lamport, 1978]* proposed the *logical time* concept that allows to abstract time. Instead of considering time (hours, minutes, ...), events at processes are considered. An *event* (also called *step*) can be an action or instruction of an algorithm such as computing instructions or sending or receiving a message. By using logical time it is possible to augment every process with a *logical clock* that allows to order events at that process.[4]

Global Clock. A discrete global clock can also be used to simplify the presentation of the model. However, no process has access to this clock; it is merely a fictional device. Events at processes are always associated with a certain global time. It is usual to assume a linear model of event execution, i.e., for every instant in time there is at most one event in the system which is executed. In this sense, the time domain T will be represented by the set of natural numbers $\mathbb{N}$.

[4] By defining *happened-before* relationships among events.

2.4.1 Synchronous Systems

A synchronous system involves systems which present very clear bounds on time. According to *[Hadzilacos and Toueg, 1994]*, a synchronous system has the following bounds:

- There are a lower and an upper time bound for every step executed by a process.
- Every sent message is received within a known bounded time.
- Every clock has a known, bounded deviation from real time.

Consensus, and therefore other agreement problems (see Section 3.3), can be solved in synchronous systems. In such systems, for example, it is possible to clearly distinguish crashed processes from *slow* processes or *slow* messages from omitted messages, which allows to deterministically detect failures.

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

So the problem of this system model is its feasibility, since it is difficult to find systems that meet these properties. Even so, if we consider the worst case (supposing very high bounds), it could be a very inefficient solution, since worst-case values are typically much higher than average values.[5]

2.4.2 Asynchronous Systems

In contrast with the previously introduced synchronous systems, now we present asynchronous systems. In an asynchronous system, also called time-free system, there is no (upper) bound on neither the transmission delay of messages nor the relative speeds of processes, i.e., there is no timing assumption.[6]

Asynchronous systems have some appealing characteristics. First, the asynchronous model fits better to unpredictable load of processes and communication channels. In this regard, the Internet is a good example of a system with those characteristics. Additionally, algorithms working in this model always preserve their correctness (safety properties) when they are executed in any other system model.

However, devising algorithms in asynchronous systems is not an easy task. When failures may occur, there are problems to deterministically distinguish *slowness* on communication (or processes) and failures, such as an omission of a message or a crash of a process. To set an example, let's suppose that a process p is waiting for a message m from another process q to know whether q has crashed or not; during its wait p does not have any way to know how long it should be waiting for the message

[5] To detect a failure it would be necessary to wait for the worst case bound time.

[6] In this kind of systems the use of the previously introduced logical time is specially interesting because it is only based on events such as message passing.

before considering that there has been a failure (either a crash of q or an omission of m). There are two possible ways of making a mistake[7]:

a) If p finally decides to consider q as faulty because the expected message has not been delivered yet, then it might happen that the expected message finally arrives, thus p made the wrong decision. Consequently, the safety of the application is violated.

b) If p decides to wait until the message is received, then it might occur that this message never gets p in case q crashed before sending it. As a result, p would wait for good. In this case the liveness of the application would not be satisfied.

There are several problems that have been proved to be impossible to solve in asynchronous systems. More specifically, we will show in Section 3.2 that consensus is impossible to solve in asynchronous systems in which a process can crash. In order to circumvent those impossibilities, some additional timing assumptions are usually assumed. In this work we will consider partially synchronous systems.

2.4.3 Partially Synchronous Systems

It is very usual that real systems hold timing bounds *most of the time*, that is, those systems behave like synchronous systems *almost always* but can be unstable during some periods of time, behaving like an asynchronous system and thus surpassing normal bounds.

partial synchrony was presented by *[Dolev, Dwork, and Stockmeyer, 1987]* in order to define an intermediate model between synchronous and asynchronous systems, namely partially synchronous systems. They proposed five parameters to classify different partially synchronous systems, and therefore 32 different models were considered. In particular, four out of those models were identified as minimal models in which consensus is solvable (in terms of synchrony).

[Dwork, Lynch, and Stockmeyer, 1988] considered two interesting partially synchronous models:

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

[7]For the sake of simplicity, we assume that only crashes may occur (no omissions).

Later, *[Chandra and Toueg, 1996]* identified the previous models as $\mathcal{M}_1$ and $\mathcal{M}_2$ and introduced another partially synchronous model, namely $\mathcal{M}_3$, with the weakest assumptions of the previous ones:

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

This partially synchronous system type will be considered in the next chapters.

In fact, it is not necessary that an eventually synchronous system holds its bounds forever, i.e., behaves like a synchronous system for good. It is sufficient if that stability lasts during the necessary time for the algorithm to finalize its execution.[8]

The Θ-model. Recently, *[Widder and Schmid, 2009]* have proposed the theta model (Θ-model), a partially synchronous timing model that assumes that, after an unknown Global Stabilization Time, the ratio between the shortest and largest time for communicating through message passing is bounded.

2.4.4 Summary

Figure 2.2 shows a representation of the main timing models presented in this section.

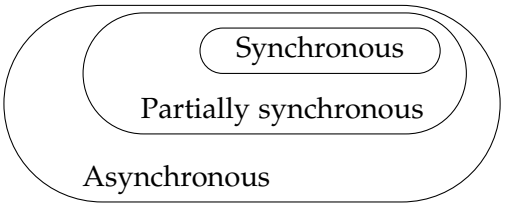


Figure 2.2: Timing models

Observe that *synchronous systems* $\subset$ *partially synchronous systems* $\subset$ *asynchronous systems*.

2.5 Process Failure Models

In Section 2.2 we introduced the *process* abstraction and commented that a process can be classified into *correct* or *faulty* depending on its behaviour; if a process suffers any kind of failure it is considered *faulty* and *correct* otherwise.[9] There are several types of failures a process can suffer. In this section we will present some failure abstractions according to the failure patterns that processes in the system can suffer. The failure abstractions we will consider in this section are *crash*, *crash-recovery*, *general omission* and *Byzantine*.

[8] Interestingly, this reasoning can also be applied to the correctness of processes.

[9] Note that later on we will define *good* and *bad* processes, definitions that do not always match *correct* and *faulty*; a *faulty* process could also be a *good* process (*correct enough*). As we will see, some failures are *benign* in the sense that they can be circumvented, in the spirit of fault tolerance.

2.5.1 Crash

In the crash failure model, also called *crash-stop* or *crash prone*, process crashes can happen. When a process suffers a crash, it stops executing its algorithm and, as a consequence, sending and receiving messages. A faulty process executes correctly before it crashes, but after it stops, it remains inactive forever, i.e., crashes are permanent and crashed processes do not recover.

Crash failures in the context of consensus were first presented in *[Lamport and Fischer, 1982]* and extended in *[Hadzilacos and Toueg, 1994]*. In *[Schlichting and Schneider, 1983]* a stronger type of crash, namely fail-stop, was defined; when a process of that type crashes, all correct processes are informed of that failure. Observe that fail-stop processes can be easily implemented in synchronous systems by using time-outs.

2.5.2 Crash-Recovery

The crash-recovery model subsumes the crash failure model and adds the possibility for a crashed process to recover. Recovered processes usually behave as if they had been reset; this implies that they *forget* everything they did/learnt until their crash, i.e., they lose all their pre-crash information, so they have to start from scratch after recovering. To avoid this problem, sometimes processes are supposed to have a *stable storage* where information can be stored and later accessed. Thus, a crashed and later recovered process could achieve essential information for correctly recovering its operational state. Since stable storage is an expensive resource, efforts have been made to use it as less as possible. Besides, instead of using stable storage, some other proposals assume that there is a minimal number of processes which do not crash and that allow recovering processes to update their information.

2.5.3 General Omission

There is another type of failure, called *omission*, which represents loss of messages while exchanging messages. This failure happens at the process, and not in the communication channel, and represents situations such as buffer overflows or malicious dropping of messages. However, according to *[Fich and Ruppert, 2003]*, omission failures can also be useful to model communication channels which lose messages.

Omissions can be classified into *send-omissions* and *receive-omissions*. A process p suffers a send-omission failure if it executes a *send* message instruction but the message never reaches the link (see Figure 2.1). A process p suffers a receive-omission when a message is received at its destination process, but at this process the message is never delivered (the *receive-message* event is never triggered). Figure 2.1 represents the difference between *receiving* and *delivering* when considering omissions.

The *send-omission* failure model was proposed by *[Hadzilacos and Toueg, 1994]* to denote a system in which processes can crash (but cannot recover) and suffer send-omissions. *[Perry and Toueg, 1986]* proposed the *general omission* failure model by adding to the previous one the possibility of suffering *receive-omission* failures as well.

Some authors distinguish permanent and transient omissions. When a process suffers a permanent omission failure, it implies that, after omitting a first message, every subsequent message will be omitted. On the other hand, a process that suffers a transient omission can reliably send/receive messages again.

2.5.4 Timing Failures

Sometimes, *timing failures* are also considered, e.g., by *[Attiya and Welch, 2004, p. 284]* or *[Coulouris, Dollimore, and Kindberg, 2005, p. 55]*. This type of failures are related to violations of bounds in synchronous systems (and therefore partially synchronous systems).[10] Timing failures are considered more severe than omissions failures but less severe than arbitrary failures.

In Chapter 6 we will present timing failures as a particular case of asynchrony due to malicious behaviour.

2.5.5 Byzantine

The *Byzantine* failure model by *[Lamport, Shostak, and Pease, 1982]* considers *arbitrary* failures, meaning that a process can make arbitrary state transitions and send arbitrary messages, deviating from the assigned algorithm. This type of failures are also called *Byzantine* or malicious.[11]

2.5.6 Connecting Failure Models

Figure 2.3, adapted from *[Barborak and Malek, 1993]* and *[Guerraoui and Rodrigues, 2006]*, depicts a classification of the previous main failure models.

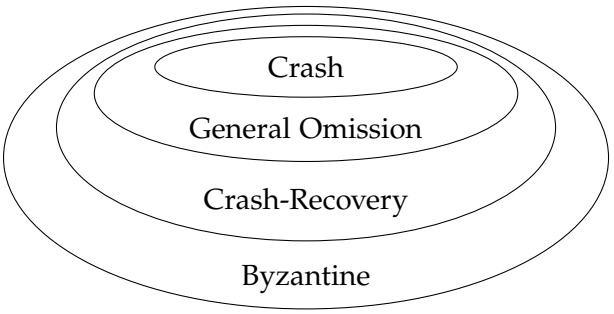


Figure 2.3: Failure models

As it can be observed *Crash* $\subset$ *General Omission* $\subset$ *Crash-Recovery* $\subset$ *Byzantine*, that is, failures in a subset are also considered in a set containing such subset. For example, *crash* failures are considered in the *Crash* failure model, but also in the rest of the

[10] Observe that timing failures can't happen in asynchronous systems, since there is no bound.

[11] Arbitrary failures can be modelled as if they were produced by a malevolent entity. It is also considered that malevolent entities at different processes can collaborate.

presented failure models. The main reason for this classification is that failures in subsets can be simulated by the containing set:

- In the general omission model, crashed processes experience the same external behaviour as processes that suffer permanent send and receive omissions.
- Omissions in the Crash-Recovery model can be simulated by instantaneous crashing and recovering at sending/receiving, using for example stable storage.
- Arbitrary failures include any kind of failure such crashes, omissions and crash-recoveries.

2.5.7 Some Remarks about the Detectability of Failures

In this work we focus on failure detectors, which are based on the detectability of failures of processes. Failure detectors have mainly been studied in the crash failure model. However, when considering other failure models, there are some interesting remarks to point out.

Remark 2.1: *Some failures are undetectable.*

To set an example, if a process p suffers send-omissions towards a crashed process, then these omissions will be undetectable and p could be considered a correct process.

Remark 2.2: *Some failures can be detected, but its type can not deterministically be identified.*

Observe that sometimes there will be no way to identify which type of failure a faulty process has suffered. If a process p does not receive any more expected messages from another process q, it could be due to the fact that, for example, q is suffering permanent send-omissions towards p. But it is also possible for q to have a Byzantine behaviour. Both cases could be undistinguishable.

Remark 2.3: *Instead of identifying* correct *processes, sometimes it is interesting to detect* good *processes.*

In the crash model, faulty processes are *not working* processes in the sense that they are not able to compute or communicate. However, in the rest of the failure models a faulty process could still be operational as long as it keeps a minimal ability to compute and communicate, even though if it is in a restricted way. This idea underlies in the concept of *good* process *[Guerraoui, Hurfin, Mostéfaoui, Oliveira, Raynal, and Schiper, 1999]*.

Sometimes we will also reference *good* processes as *correct enough*.

Chapter 3

Consensus

Consensus is one of the most important problems in asynchronous systems (Section 2.4.2) and a paradigm which represents a family of agreement problems (*[Pease, Shostak, and Lamport, 1980; Lamport et al., 1982; Turek and Shasha, 1992; Barborak and Malek, 1993]*). In addition to the importance of the problem itself, many practical agreement problems can be solved by reducing them to the consensus problem (see Section 3.3). In this sense, consensus can be used as a building block to solve some other problems. In Chapter 7 we will present an example of such functionality.

Outline. Section 3.1 presents an introduction to consensus, describing its primitives and properties. Section 3.2 addresses the feasibility of consensus in synchronous and asynchronous systems. Finally, Section 3.3 presents some related problems to consensus.

3.1 Overall Description

As stated above, this problem can be considered as a paradigm of agreement; processes *propose* an initial value (from a fixed alphabet) and later have to *decide* by agreeing on one of the proposed values (*[Chandra and Toueg, 1996]*). Entering into detail, consensus is based on two primitives:

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

Consensus has trivial solutions in systems in which processes do not fail. However, reaching agreement in systems in which processes can fail can be much harder. *[Pease et al., 1980]* considered faulty processes, which could make agreement more difficult. *[Lamport et al., 1982]* introduced the *Byzantine Generals Problem*, also called *Byzantine agreement*, in which several generals or lieutenants (processes) receive a command (*attack* or *retreat*) and all of them have to agree on the same action. Moreover, there may

be *traitors* who try to mix up the rest of generals. Finally, the decided action has to be the same as the initial command if the commander who issued it was *loyal*. Although this problem is considered a paradigm of the consensus problem, in this case different roles are distinguished (commander, general).

For the sake of simplicity, in this chapter we will assume that processes communicate through reliable message passing, i.e., sent messages are eventually received and cannot be modified nor duplicated.

Properties

Solutions to the consensus problem must guarantee the following properties, according to *[Chandra and Toueg, 1996]*:

- C-Termination. Every *correct* process eventually decides some value.
- C-Agreement. Not two correct processes decide differently.
- C-Validity. If a process decides a value v, then v was previously proposed by some process.
- C-Integrity (or irrevocability). No process decides twice.

These properties can be classified according to safety and liveness, introduced in the previous chapter (Section 2.1); C-Agreement, C-Validity and C-Integrity are safety properties, whereas C-Termination is a liveness property.

Uniform consensus. According to the definition of the C-Agreement property of consensus, a faulty process may decide a different value from the one that correct processes agree on. In *uniform* consensus all deciding processes, correct or faulty, agree on the same value. Uniform consensus redefine C-Agreement as:

- Uniform agreement. No two processes decide differently.

The consensus algorithms proposed in this work also solve uniform consensus.

3.2 Consensus and Timing Models

In this section we study the feasibility of consensus in synchronous and asynchronous systems, both of them already presented in Section 2.4. There are many proposals for solving in synchronous systems. However, when considering asynchronous systems there is an impossibility result of consensus in case that at least one process can crash, which has led to look for alternatives to solve consensus in such systems.

In this section we will slightly present consensus in synchronous systems and then focus on asynchronous systems.

Consensus in synchronous systems

Consensus is deterministically solvable in synchronous systems. If we assume lower and upper bounds on process speeds and communication delays then consensus can be achieved despite the crash of any number of processes (*[Lynch, 1996]*).

Specifically there are some interesting results regarding the minimal number of rounds of exchanged messages (sometimes these rounds are also called *shots*) or the minimal number of correct processes needed to achieve consensus. In *[Lamport et al., 1982; Fischer and Lynch, 1982]* it was shown that in a system with f faulty processes the lower bound to achieve consensus is $f + 1$ rounds (and only one round if there is no failure). Additionally, *[Lamport et al., 1982; Pease et al., 1980]* showed that in a system in which arbitrary failures can happen, a minimal number of correct processes is required being $c > 2f$,[1] where c is the number of correct processes and f the number of the faulty ones, as introduced in Section 2.2.

Further description can be consulted in *[Attiya and Welch, 2004; Coulouris et al., 2005; Guerraoui and Rodrigues, 2006; Raynal, 2010]*.

Consensus in asynchronous systems where a single process can fail

[Fischer et al., 1985] proved that consensus can not deterministically be solved in an asynchronous system if a process may crash. This impossibility result, also called *FLP*, has had a great impact on this area, since it implies that there are also some other classical problems in distributed computing in crash-prone asynchronous systems which are impossible to solve.

Specifically, *[Fischer et al., 1985]* showed that it is not possible to fulfil reliable failure detection in asynchronous systems due to the inability to distinguish a crashed process from a *slow* process (assuming that processes only fail by crashing). If a process p is waiting for an expected message m from another process q (recall that we assume reliable channels and processes omit no message), this question arises; how long should p wait for m? If p gives up waiting and considers q as faulty, safety could be violated (in case m could arrive a bit later). However, if p decides to wait forever, liveness could be violated if q crashed before sending m (so m will never be received by p). To summarize, guaranteeing termination properties is not possible in asynchronous systems. For further information, *[Turek and Shasha, 1992; Freiling and Völzer, 2006; Herlihy, 2008]* illustrate the FLP impossibility result.

Observe that, since FLP was proved for the crash failure model, it is also applicable to the rest of failure models presented in section 2.5.

Note also that FLP affects many other problems which are related to consensus, such as Atomic Commitment (how long should processes wait for another process before aborting?) and some others (as shown in 3.3).

[1] or $n > 3f$, being n the number of processes in the system.

3.2.1 Circumventing FLP

As stated before, the FLP impossibility result showed that consensus can not be achieved in pure asynchronous systems. However, solutions can be found with additional assumptions. In this section we present some proposals:

Partial synchrony. Partial synchrony was introduced in Section 2.4.3. In this case, systems become synchronous eventually, so, instead of using mere timeouts, algorithms should implement adaptive timeouts. *[Dolev et al., 1987]* considered 32 partially synchronous systems and showed that 4 out of those 32 models are minimal to solve consensus. *[Dwork et al., 1988]* considered two partially synchronous models in which consensus is solvable, proved that consensus is solvable as long as $f < n/2$ and proposed some *eventually synchronous* consensus algorithms. *[Chandra and Toueg, 1996]* considered another partially synchronous model even weaker than the previous two ones of *[Dwork et al., 1988]*, and showed that consensus is solvable in that model as well.

Failure detectors. A failure detector is an abstract module, available at every process, that informs in a possibly unreliable way about the operational state of processes in the system. Implementations of such a module may differ depending on the considered system model. We will describe failure detectors in more detail in Chapter 4.

Randomization. *[Ben-Or, 1983]* proposed a randomized solution to consensus. However, this approach is not deterministic, so we will not consider it.

3.3 Consensus and Related Problems

As stated in the introduction, consensus can be used as a building block to solve some other problems related to distributed systems. In this section we will introduce some of these problems.

3.3.1 *k*-set Agreement

The k-*set agreement* problem was initially proposed by *[Chaudhuri, 1993]* and is considered as a generalization of the consensus problem; instead of agreeing on a unique value, processes are allowed to decide among k different values. Note that consensus and k-set agreement are equivalent when $k = 1$. It is also generally assumed that the number of processes that can crash in the system is at least equal or bigger than the number of choices, since otherwise the solution is trivial.

Although several solutions to this problem have been proposed for synchronous systems, *[Borowsky and Gafni, 1993; Herlihy and Shavit, 1999; Saks and Zaharoglou, 2000]* showed that it is impossible to deterministically solve this problem in asynchronous systems when the number of faults is equal or bigger than k; observe that if there is only one faulty process and $k = 1$ then the impossibility result of *[Fischer et al., 1985]* can also be applied. As in consensus, several ways have been proposed to circumvent this impossibility result, such as randomization or failure detectors.

3.3.2 Atomic Broadcast, also Totally Ordered Broadcast

In the *Atomic Broadcast* problem, processes have to agree on a set of messages and a delivery order for those messages. Introduction and survey on algorithms addressing Atomic Broadcast can be found in *[Défago, Schiper, and Urbán, 2004; Défago, 2008]*. *[Chandra and Toueg, 1996]* showed that consensus and Atomic Broadcast are equivalent in asynchronous crash-prone systems, i.e., Atomic Broadcast can be reduced to consensus and vice versa.

3.3.3 Non-Blocking Atomic Commitment (NB-AC)

Atomic Commitment protocol (AC) ensures the *all-or-nothing* property of transactional systems; every process *votes* an initial value, which can be *commit* or *abort*; if all processes vote to commit and no process fails, then committing should be decided, otherwise, aborting.

The two-phase commit protocol (2PC) (*[Gray, 1978]*) solves AC, but it can block in presence of faults. To solve this problem, *Non-Blocking Atomic Commitment* (NB-AC)[2] was proposed in order to ensure the termination of transactions in distributed databases.

Both consensus and NB-AC are solvable in synchronous systems, e.g., in the case of NB-AC by the three-phase commit protocol (3PC) *[Skeen, 1981; Keidar and Dolev, 1995]*. Nevertheless NB-AC is not solvable in some non-synchronous systems in which consensus is solvable,[3] so it can be stated that NB-AC is a harder problem than consensus is in such systems. In fact, NB-AC is not solvable in the partially synchronous systems proposed by *[Dwork et al., 1988]*, as shown by *[Guerraoui, 1995; Charron-Bost, 2003]*.

[Guerraoui, 1995] presented the *Non-Blocking Weak Atomic Commitment* problem (NB-WAC) by redefining the *Non-Triviality* condition of NB-AC; if all participants vote *yes* and no participant is ever suspected, then every correct participant eventually decides *commit*. It was shown that NB-WAC can be reduced to consensus. *[Guerraoui, Larrea, and Schiper, 1995]* proposed an algorithm solving NB-WAC by using failure detectors.

3.3.4 Uniform Reliable Broadcast

Uniform Reliable Broadcast, URB, is a communication primitive which satisfies the following property: if a process delivers a message, then all correct processes also deliver this message. It was first proposed by *[Hadzilacos and Toueg, 1994]*.

Formally, Uniform Reliable Broadcast must satisfy the following properties:

- Validity. If a correct process broadcasts a message m, then it eventually delivers m.

[2] AC where correct participants terminate despite faults of participants.

[3] Recall that in asynchronous systems neither of them is solvable due to the impossibility result of *[Fischer et al., 1985]*.

- Uniform agreement. If a process (whether correct or faulty) delivers a message m, then all correct processes eventually deliver m.
- Uniform integrity. For any message m, every process (whether correct or faulty) delivers m at most once, and only if m was previously broadcast by the sender of m.

Chapter 4

Failure Detectors

Failure detectors were proposed by *[Chandra and Toueg, 1996]* as an abstraction to encapsulate timing assumptions when solving consensus. In this chapter we present an introduction to failure detectors. For a more pedagogic introduction and further information, the following works can be consulted: *[Mostéfaoui, Mourgaya, and Raynal, 2002; Raynal, 2005; Guerraoui and Rodrigues, 2006; Guerraoui, 2008; Raynal, 2010; Freiling, Guerraoui, and Kouznetsov, 2011]*.

Outline. In this chapter we introduce and formalize failure detectors, focusing on the crash failure model. Section 4.1 introduces failure detectors informally. Then, in Section 4.2 formalization is included. Section 4.3 defines the main properties of a failure detector and an initial classification according to these properties is presented in Section 4.4. Finally, Section 4.5 shows how failure detector classes can be compared.

4.1 General Description

Roughly speaking, a failure detector is an abstract module located at each process in the system that provides (possibly unreliable) information about (the operational state of) other processes in the system. One of the main objectives of this module is to provide an abstraction of the timing assumptions of the system —i.e., failure detectors encapsulate the synchrony of the network—, so that applications on top do not have to worry about these assumptions, so they can be designed as if they run in an asynchronous system.

Failure detectors are assumed to provide *unreliable* information because they can make false suspicions, e.g., by suspecting a non-faulty process or not suspecting a faulty process. Failure detectors reliability will be formalized through two properties: *completeness* and *accuracy* (see Section 4.3). Observe that the information provided by failure detector modules can differ from one process to another.

The concept of a failure detector has been investigated in quite some detail in systems with merely crash failures (*[Freiling et al., 2011]*). In such systems, correct processes

(i.e., processes which do not crash) must eventually and permanently suspect crashed processes.

Failure detectors have been used to solve several problems in asynchronous crash-prone distributed systems, in particular the *consensus* problem, which we introduced in Chapter 3.

Figure 4.1 presents the layers in a process when using a failure detector to solve consensus. The *Transport* layer provides processes with the ability to send and receive messages through the communication channel. The *Failure Detector* layer monitors other processes and offers possibly *unreliable* information about faulty processes to the upper layer. The *Consensus* layer uses information from its failure detector module in order to achieve agreement among correct processes. Finally, the *Application* layer represents the set of potential applications which could profit from the consensus service.

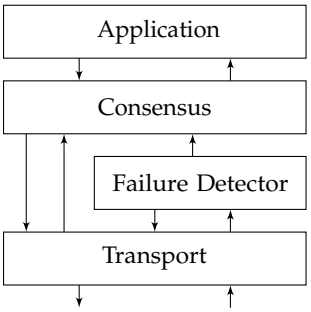


Figure 4.1: Layers in a process when using a failure detector to solve consensus

As stated above, a failure detector provides information about the operational state of other processes. *[Chandra and Toueg, 1996]* proposed failure detectors that output a list of suspected processes. *[Hutle, 2005]* denotes the previous type of failure detectors as *suspicion based* and additionally characterizes as *trust based* those failure detectors which output a list of non-*suspected* processes, i.e., *trusted* processes. To set an example, the eventually perfect failure detector, denoted $\diamond\mathcal{P}$, is a *suspicion based* failure detector, whereas the eventual leader failure detector, denoted Ω, a *trust based* one.

4.2 Formal Definitions

Failure detectors were first proposed for the crash failure model, already presented in Section 2.5.1. In this section we also assume such failure model, whereas in Chapter 6 we will consider the general omission failure model.

Failure Pattern. A crash failure pattern is a function that defines a possible set of failures, and their corresponding time, that can occur in an execution. This function can be formalized as $F_{pa} : T \rightarrow 2^{\Pi}$, where $F_{pa}(t)$ denotes the set of processes that have crashed up to time t of an assumed global clock (see Section 2.4). Observe that

$F_{pa}(t) \subseteq F_{pa}(t+1)$.[1] We will consider F as the set of processes that will crash during an execution, i.e., $F = F_{pa}(t_{max})$, being t_{max} the time when the execution finishes. In the same way, we consider the set of correct processes defined as $C = \Pi - F$. Formally:

Observation 4.1:

$$\forall F_{pa}, \exists t \in T, t \leq t_{max} : \\ (F_{pa}(t) = F) \wedge (\Pi - F_{pa}(t) = C) \wedge (\forall t' > t : F_{pa}(t') = F_{pa}(t))$$

Failure Detector History. Following the definition proposed in *[Chandra and Toueg, 1996]*, a failure detector history is a function that provides a list of the processes that a failure detector is suspecting at a given time. Formally a failure detector history H is a function $H : \Pi \times T \mapsto \mathcal{R}$, where *range* $\mathcal{R} = 2^{\Pi}$. For example, $H(p_1, t) = \{p_2, p_4\}$ means that at time t process p_1 considers processes p_2 and p_4 as faulty.

Failure Detector. We define a failure detector $\mathcal{D}$ as a function that maps a failure pattern F_{pa} to a set of failure detector histories $\mathcal{D}(F_{pa})$. Intuitively, these are all possible outputs, i.e., failure detector histories, which could be given in executions with failure pattern F_{pa} and failure detector $\mathcal{D}$. Each history represents a possible behaviour of $\mathcal{D}$ for the failure pattern F_{pa}. Every process p_i has a local failure detector module of class $\mathcal{D}$ denoted $\mathcal{D}_i$.

Failure Detector Class. Failure detectors can be classified into classes based on properties they satisfy (see next section). Failure detector classes compose an interesting abstraction when looking for solutions for a given problem P, e.g., consensus (see Chapter 3), in a given system model S. Recall that different possibilities for defining a system model were already presented in Chapter 2.

Defining failure detectors in terms of their class allows applications to make use of them regardless of their implementation. In this way, an application can replace its failure detector adapted to an specific system model with another failure detector which provide the same properties in another system model. As a result, the application works in both system models without any changes. This is one of main advantages of defining failure detector classes.

From now on we will refer to a *failure detector class* as failure detector (and not for a particular failure detector implementation) and denote it as $\mathcal{D}$.

4.3 Properties

[Chandra and Toueg, 1996] classified failure detectors according to two abstract properties: *completeness* and *accuracy*. While completeness characterizes the failure detector capability of suspecting faulty processes, accuracy restricts the mistakes the failure detector can make. Both properties are complementary. For example, if we only consider

[1] Note that crashed processes do not recover.

completeness, it could be easily achieved by suspecting every process in the system. In the same way, accuracy would be met if there is no suspicion at all. Thus, neither of both is useful separately, so both need to be taken into account at once.

4.3.1 Completeness

[Chandra and Toueg, 1996] proposed two completeness properties that a failure detector $\mathcal{D}$ can satisfy $\forall F_{pa}, \forall H \in \mathcal{D}(F_{pa})$:

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

$$\exists t \in T, \forall p \in F, \forall q \in C, \forall t' \geq t : p \in H(q, t')$$

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

$$\exists t \in T, \forall p \in F, \exists q \in C, \forall t' \geq t : p \in H(q, t')$$

Strong completeness guarantees that every correct process will eventually and permanently suspect every faulty process. On the other hand, weak completeness guarantees that every faulty process will be eventually and permanently suspected by at least one correct process.

4.3.2 Accuracy

In *[Chandra and Toueg, 1996]*, some accuracy properties were also proposed. A failure detector $\mathcal{D}$ can satisfy $\forall F_{pa}, \forall H \in \mathcal{D}(F_{pa})$:

- Strong Accuracy. No process is suspected before it crashes.

$$\forall t \in T, \forall p, q \in (\Pi - F_{pa}(t)) : p \notin H(q, t)$$

- Weak Accuracy. Some correct process is never suspected

$$\exists p \in C, \forall t \in T, \forall q \in (\Pi - F_{pa}(t)) : p \notin H(q, t)$$

Observe that strong accuracy means that no correct process can ever be suspected. This implies that every suspected process is a faulty process. For this reason, failure detectors which satisfy strong accuracy are mean to *detect* faulty processes, instead of merely *suspecting*.

Both weak and strong accuracy properties are very hard to achieve, since there has to be at least one process which is never suspected, so the following more relaxed property was also proposed in *[Chandra and Toueg, 1996]*.

Eventual accuracy.

- Eventual Strong Accuracy. There is a time after which correct processes are not suspected by any correct process.

$$\exists t \in T, \forall t' \geq t, \forall p, q \in (\Pi - F_{pa}(t)) : p \notin H(q, t')$$

  By observation 4.1 (page 25):

$$\exists t \in T, \forall t' \geq t, \forall p, q \in C : p \notin H(q, t')$$

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

$$\exists t \in T, \exists p \in (\Pi - F_{pa}(t)), \forall t' \geq t, \forall q \in (\Pi - F_{pa}(t)) : p \notin H(q, t')$$

  By observation 4.1:

$$\exists t \in T, \exists p \in C, \forall t' \geq t, \forall q \in C : p \notin H(q, t')$$

In contrast with the observation made when presenting strong accuracy, Eventual Strong Accuracy ensures that a failure detector which fulfils such property will *eventually detect* only faulty processes, that is, eventually there will be no false suspicion.

4.4 Classical Failure Detector Classes

According to the properties introduced in the previous section, *[Chandra and Toueg, 1996]* proposed some failure detector classes. Since a failure detector is characterized by a completeness and by an accuracy property, the combination of these properties offers us eight classes of failure detectors, which are shown in Table 4.1.

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

Table 4.1: Eight classes of failure detectors

In addition to the previous classes, *[Chandra, Hadzilacos, and Toueg, 1996]* introduced another failure detector class, denoted Omega, Ω , which provides eventual agreement on a common and correct *leader* among all non-faulty processes in a system. Observe

that Ω is a *trust based* failure detector, whereas the previous ones are *suspicion based* failure detectors.

$$\exists t_1 \in T, \exists p \in C, \forall t_2 \geq t_1, \forall q \in C : H(q, t_2) = \{p\}$$

where failure detector history $H(q, t)$ provides the *leader* which q trusts at time t.

The *Perfect* failure detector $\mathcal{P}$ is reliable in the sense that every suspected process is a faulty process, i.e., $\mathcal{P}$ detects faulty processes. On the other hand, $\diamond\mathcal{P}$ is unreliable since it can make mistakes, even though eventually it will make no mistake. *[Guerraoui, 2000]* showed that algorithms devised to rely on unreliable failure detection are *indulgent* algorithms, meaning that they never violate safety and that they achieve *uniform* solutions, i.e., all processes that terminate the algorithm agree, regardless if they are correct or faulty-but-still-up.

4.5 Reducibility

In this section we will introduce the relation *weaker than* for comparing failure detectors. A failure detector $\mathcal{D}_1$ is weaker than another failure detector $\mathcal{D}_2$, denoted $\mathcal{D}_1 \preceq \mathcal{D}_2$, if there is an asynchronous algorithm which can emulate $\mathcal{D}_1$ by using $\mathcal{D}_2$. It is also said that $\mathcal{D}_2$ is *reducible* to $\mathcal{D}_1$ by a *reduction* algorithm.

- If $\mathcal{D}_1 \preceq \mathcal{D}_2$ and $\mathcal{D}_2 \preceq \mathcal{D}_1$ then $\mathcal{D}_1 \equiv \mathcal{D}_2$ ($\mathcal{D}_1$ and $\mathcal{D}_2$ are *equivalent*).
- If $\mathcal{D}_1 \preceq \mathcal{D}_2$ and $\mathcal{D}_2 \not\preceq \mathcal{D}_1$ then $\mathcal{D}_1 \prec \mathcal{D}_2$ ($\mathcal{D}_1$ is *strictly-weaker* than $\mathcal{D}_2$).

Some interesting remarks about reducibility:

- A reduction algorithm is assumed to be asynchronous and as a consequence it does not implement any function regarding failure detection.
- A reduction algorithm often implements a distributed variable with the output of the emulated failure detector.
- A reduction algorithm depends on the system model. This implies that the *weaker than* function is only valid for the considered system model.

The weakest failure detector problem. The ability to compare failure detectors leads us to look for the *weakest* failure detector to solve a given problem P. Formally, a failure detector $\mathcal{D}$ is the weakest failure detector to solve P if:

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

In this regard, *[Chandra et al., 1996]* showed that Ω is the weakest failure detector for solving consensus with a majority of correct processes.

Interestingly enough, the weakest failure detector problem turns into a way to compare and classify problems. A problem P_1 is *harder* than another problem P_2 if the weakest failure detector to solve P_2 is weaker than the weakest failure detector to solve P_1. Recall that the *weaker than* function depends on the system model under consideration, so the problem classification by failure detectors also depends on a specific system model.

[Aguilera, Toueg, and Deianov, 1999; Delporte-Gallet, Fauconnier, Guerraoui, Hadzilacos, Kouznetsov, and Toueg, 2004; Jayanti and Toueg, 2008] study the *weakest failure detector* for various problems.

4.6 Implementation Approaches

Failure detectors are abstract devices, i.e., they are not attached to a specific implementation. Therefore, the implementation of a failure detector is open as long as it meets the properties of the desired class.

Traditionally, failure detectors are based on monitoring other processes. There are mainly two methods to implement such monitoring: *polling* and heartbeat sending.

Polling. (or *query/reply*) When a process p monitors another process q by polling, p sends a query message to q and stays waiting for an answer to this message from q. If p does not receive an answer after a given time, p will suspect q, i.e., p will consider q as faulty. For example, failure detectors based on polling have been proposed in *[Larrea, Arévalo, and Fernández, 1999; Larrea, Fernández, and Arévalo, 2004]*.

Heartbeat. In this approach, every monitored process q sends periodic heartbeat messages to monitoring processes in order to inform them that it is still *alive*. If monitoring processes do not receive expected messages from q after a given time, they will suspect q.

Observe that in both cases the waiting time depends on the timing of the system model (see Section 2.4); if the system is synchronous, that time is known and can be set beforehand. However, if the system is partially synchronous, a mechanism to adjust that time is necessary. In this regard, the time-outs based approach is one of the most followed approaches.

Communication pattern. Failure detectors implementations can follow different communication patterns. This is an important issue when considering efficiency in communication. For example, an all-to-all communication pattern presents some advantages, such as responsiveness, but lacks efficiency in terms of communication. On the other hand a linear communication pattern would be desirable to achieve a higher degree of communication efficiency. In Chapters 5 and 8 we will study the effect of the communication pattern on the communication efficiency.

4.7 Solving Consensus with Failure Detectors

[Chandra and Toueg, 1996] showed that $\diamond\mathcal{S}$ is the weakest class of failure detectors allowing to solve consensus, assuming a majority of correct processes and proposed a centralized consensus algorithm based on a $\diamond\mathcal{S}$ failure detector. This algorithm can be found in Appendix A and follows a round-based, rotating coordinator paradigm. *[Hurfin and Raynal, 1999]* and *[Schiper, 1997]* also presented algorithms following this approach. *[Guerraoui et al., 1999]* present a comparison of those three algorithms.

Chapter 5

Communication Efficiency in the Crash Failure Model

Failure detectors, which were already presented in Chapter 4, have been mainly studied in the crash model. This chapter looks into the implementation of failure detectors in such a failure model searching eventual *communication efficiency*, that is, trying to minimize the number of links that eventually will keep sending messages. More specifically, we mainly focus on the class $\diamond\mathcal{P}$ of failure detectors. Although other failure detector classes such as $\diamond\mathcal{S}$ or Ω can also be used to solve consensus, $\diamond\mathcal{P}$ is stronger and hence it provides better accuracy. We define communication optimality, which improves communication efficiency and present some communication-optimal implementations of the $\diamond\mathcal{P}$ failure detector class. Finally, we compare different $\diamond\mathcal{P}$ failure detector class implementations in terms of communication efficiency.

Outline The chapter is organized as follows. Section 5.1 presents a little background regarding the subject of this chapter. In Section 5.2, we describe the system model considered in this chapter. Section 5.3 introduces communication efficiency, whereas Section 5.4 presents *communication optimality*, a measure that improves communication efficiency. Section 5.5 proposes two communication-optimal $\diamond\mathcal{P}$ algorithms (Sections 5.5.1 and 5.5.2). In Section 5.5.3, we analyse the performance of those algorithms, comparing them with some other previously proposed $\diamond\mathcal{P}$ algorithms. Finally, Section 5.6 concludes the chapter.

5.1 Related work

Failure detectors provide information about other processes *correctness*, so they need a inherent mechanism to periodically monitor other processes in the system. Depending on the chosen mechanism for implementing that monitoring process, communication costs in terms of the number of messages being sent can be very different. We will focus on two parameters considered when implementing failure detectors: monitoring mechanisms, e.g., heartbeat or polling and communication patterns, e.g., *all-to-all* (recall that those mechanisms were previously introduced in Section 4.6).

The algorithm proposed in the failure detector seminal paper (*[Chandra and Toueg, 1996]*) uses a heartbeat mechanism and all-to-all communication to detect faulty processes. The algorithms proposed by *[Aguilera, Delporte-Gallet, Fauconnier, and Toueg, 2001]* and by *[Larrea, Fernández, and Arévalo, 2005]* use heartbeats too, but rely on a leader-based approach. On the other hand, the algorithms proposed by *[Larrea et al., 1999 and 2004]* use a polling —or query/reply— mechanism on a ring arrangement of processes. Roughly speaking, two conclusions can be drawn; first, the leader-based and the ring-based algorithms are more efficient than the all-to-all algorithm regarding the number of messages sent (linear vs. quadratic). Second, observe also that, compared to polling, the heartbeat mechanism reduces the number of messages to the half. Moreover, heartbeats inherently provide a certain level of communication reliability in a system with fair lossy links, while polling usually requires reliable communication (both options were introduced in Section 2.3).

In this chapter we consider a partially synchronous model, which was introduced in Section 2.4.3. Recall that in such a model, in every run of the system, there are bounds on relative process speeds and on message transmission times, but these bounds are not known and they hold only after some unknown but finite time. In practice, the bounds depend on parameters such as the network speed or the size of the system, and hold easier in, for example, local area networks than in wide area networks. Hence, it is important to design failure detection algorithms that use a low number of links. In this sense, *[Aguilera et al., 2001]* defined *communication efficiency* for algorithms that eventually use only n unidirectional links. With regard to this performance measure, heartbeat-based ring algorithms outperform other ring algorithms based on polling *[Larrea et al., 1999 and 2004]* or algorithms using a centralized communication pattern *[Aguilera et al., 2001; Larrea et al., 2005]*. By all means, algorithms using an all-to-all communication pattern, such as the algorithm in *[Chandra and Toueg, 1996]*, are far from being communication-efficient.

Other failure detection algorithms, specifically designed for wide area networks, e.g., *[Hutle, 2005]*, are based on failure detectors that provide their properties in a neighborhood of processes (using all-to-all communication inside each neighborhood), and propagate the information about suspicions among neighborhoods. The advantage of a ring based approach is that, once the logical ring is defined, neighborhoods are implicit, and eventually involve just two processes for every correct process, i.e., its correct predecessor and correct successor in the ring.

Using $\diamond\mathcal{P}$ to Solve Consensus

Despite $\diamond\mathcal{S}$ and Ω have been extensively used to solve consensus *[Chandra and Toueg, 1996; Schiper, 1997; Hurfin and Raynal, 1999; Mostéfaoui and Raynal, 1999 and 2001; Aguilera et al., 2001; Aguilera, Delporte-Gallet, Fauconnier, and Toueg, 2003 and 2004]*, we focus our work on implementing communication-efficient failure detectors of the class $\diamond\mathcal{P}$. The choice of $\diamond\mathcal{P}$ is mainly justified by the fact that, as we will also show, communication efficiency is almost the same for $\diamond\mathcal{P}$, $\diamond\mathcal{S}$ and Ω, and a failure detector

of the class $\diamond\mathcal{P}$ trivially implements $\diamond\mathcal{S}$ and Ω. Hence, consensus algorithms based on either $\diamond\mathcal{S}$, Ω, or an equivalent leader election mechanism, e.g., *[Lamport, 1998; Greve, Hurfin, Macêdo, and Raynal, 2000; Guerraoui and Raynal, 2004; Larrea et al., 2005]*, can also benefit of communication-efficient implementations of $\diamond\mathcal{P}$. Moreover, for certain problems *[Guerraoui, Kapalka, and Kouznetsov, 2006]* and consensus protocols *[Wu, Cao, Yang, and Raynal, 2006]* failure detector $\diamond\mathcal{P}$, being stronger than $\diamond\mathcal{S}$ and Ω, is required. Finally, failure detectors of the class $\diamond\mathcal{P}$ are more natural, in the sense that all the correct processes can provide a precise set composed of exclusively crashed processes, providing a better degree of accuracy.

5.2 System Model

5.2.1 Processes and Links

According to the concepts presented in Chapter 2, we consider a distributed system composed of a finite set Π of $n > 1$ processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving messages. We also use the alternative notation p, q, r, ... to denote processes. Every pair of processes (p, q) is connected by two unidirectional and reliable logical communication links c_{pq} and c_{qp}.

We consider that processes are arranged in a logical ring. Without loss of generality, process p_i is preceded by process p_{i-1}, and followed by process p_{i+1}. As usual, p_1 follows p_n in the ring. In general, we will use the functions $pred(p)$ and $succ(p)$ respectively to denote the predecessor and the successor of a process p in the ring.

5.2.2 Failure Model

We assume the crash failure model, already presented in Section 2.5.1, in which processes can only fail by crashing, that is, by prematurely halting. Moreover, crashes are permanent, i.e., crashed processes do not recover. In every run of the system we identify two complementary subsets of Π: the subset of processes that do not fail, denoted *correct* or C, and the subset of processes that do fail, denoted *crashed* or F (faulty). We use c to denote the number of correct processes in the system in the run of interest, which we assume is at least one, i.e., $c \geq 1$, where $c = |C|$.

5.2.3 Time Model

Concerning timing assumptions, we consider a partially synchronous model (introduced in Section 2.4.3), which stipulates that, in every run of the system, there are bounds on relative process speeds and on the message transmission time, but these bounds are not known and they hold only after some unknown but finite time (called *GST* for *Global Stabilization Time*). The communication links supporting this behaviour are also called *eventually timely* links *[Aguilera et al., 2003]*. Our model is actually a variant of the partial synchrony models of *[Dwork et al., 1988; Chandra and Toueg, 1996]*. The difference is that we assume reliable communication links. Nevertheless, when

presenting the algorithms we will refine the minimal assumptions on communication reliability and synchrony required by each algorithm.

Finally, in the algorithms presented in this chapter we assume that a local clock that can measure real-time intervals is available to each process. Clocks are not synchronized.

5.3 Communication Efficiency

Failure detector performance can be considered according to several measures. In this work we will mainly focus on *communication efficiency*. *[Aguilera et al., 2001]* defined communication efficiency for the Ω failure detector implementations in partially synchronous systems with the crash failure model; an algorithm for Ω is communication-efficient if it eventually uses only n unidirectional links, where n is the number of processes. That is to say that eventually only n links will be carrying messages forever.

In *[Larrea, Lafuente, Soraluze, Cortiñas, and Wieland, 2007b]*, we propose some communication-efficient algorithms for the $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$ classes. These failure detector implementations follow a heartbeat-based detection mechanism on a logical ring arrangement of processes. One of those algorithms has been included in Appendix C as a reference.

5.4 Communication Optimality

We define communication optimality to mean that only c unidirectional links carry messages forever, being c the number of correct processes.

Figure 5.1 illustrates the difference between communication efficiency and communication optimality when processes are arranged according to a ring topology. The figure shows an example of the number of unidirectional links used permanently for a system composed of eight processes, out of which five are correct, i.e., $n = 8$ and $c = 5$. Faulty processes are represented by grey circles. Observe that in a communication-efficient algorithm, e.g., *[Larrea et al., 2007b]*, n links are used permanently,[1] while in a communication-optimal algorithm only c links are used permanently, which is optimal for implementing $\diamond\mathcal{P}$.

Now we will show that c, i.e., the number of correct processes in the system, is the minimum number of unidirectional links carrying messages forever necessary for an algorithm to provide the properties of $\diamond\mathcal{P}$. Then, we show that, assuming that at least one process crashes, i.e., $c < n$, c is also minimal for implementing Ω.[2] Both results hold when there are at least two correct processes in the system, i.e., $c \geq 2$.

Theorem 5.1: c *is the minimum number of unidirectional links carrying messages forever necessary for an algorithm to provide the properties of $\diamond\mathcal{P}$ in a crash-prone system.*

[1] In the algorithm in *[Larrea et al., 2007b]*, for every process (correct or faulty), the link coming from its correct predecessor in the ring is used permanently.

[2] By equivalence, the reasoning regarding Ω applies to $\diamond\mathcal{S}$ too.

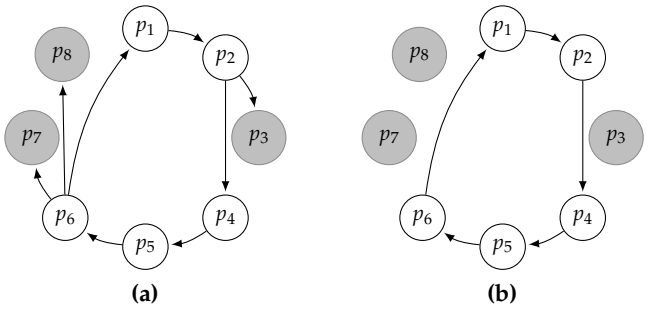


Figure 5.1: Links used permanently in (a) a communication-efficient algorithm, and (b) a communication-optimal algorithm

Proof. Given a run R, observe that every process must periodically inform that it is still alive by sending a message, which after every incorrect process has crashed gives us the minimum number of c unidirectional links. Otherwise, if less than c unidirectional links carry messages forever, there is some correct process p that eventually stops sending messages. Let t be the time instant in which p stops sending messages. Consider now another run R', identical to R until time t, and assume that p crashes at time t in R'. For any correct process q, if q does not eventually and permanently suspect p, then the strong completeness property of $\diamond\mathcal{P}$ is violated. Hence, q will eventually and permanently suspect p in R'. Observe that both executions R and R' are indistinguishable. Hence, in run R q will also eventually and permanently suspect p, violating the eventual strong accuracy property of $\diamond\mathcal{P}$. □

Theorem 5.2: *If at least one process crashes, then* c *is the minimum number of unidirectional links carrying messages forever necessary for an algorithm to provide the property of Ω in a crash-prone system.*

Proof. The proof is by contradiction. Assume that we have an implementation of Ω in which only $c-1$ unidirectional links carry messages forever. Observe that such an implementation will be possible only if correct processes are arranged in a tree topology, being the leader the root of the tree and propagating heartbeat messages —directly or indirectly— to the rest of correct processes. Consider a run R of the algorithm in which c processes are correct and let t be the time instant after which only $c-1$ unidirectional links carry messages forever. Consider now another run R', identical to R until time t, and assume that a process q different from the leader, which is correct in R, crashes at time t in R'. Observe that both executions R and R' are indistinguishable, and there is no way for the leader to know that q has crashed, and hence it will not stop sending messages to q. Since the number of correct processes in run R' is $c-1$, the algorithm should use only $c-2$ unidirectional links to carry messages forever, which contradicts the fact that the leader will not stop sending messages to q. □

[Aguilera et al., 2004] propose an algorithm implementing Ω such that eventually only f links carry messages forever, being f the maximum number of processes that can crash. They also show that in the crash failure model no algorithm using fewer than f links exists. Hence, if $f = n - 1$ (as in our system model), Ω can be implemented with $n - 1$ links carrying messages forever, even if no process crashes, i.e., $c = n$. However, the algorithm of *[Aguilera et al., 2004]* uses always $n - 1$ links, independently of the actual number of correct processes c. As we will see, the algorithms proposed in this chapter, besides implementing $\diamond\mathcal{P}$, dynamically adapt the number of links used forever to the actual number of correct processes.

Similar results can be deduced from the work by *[Fernández, Jiménez, and Arévalo, 2006]*. They study the minimal system conditions to implement unreliable failure detectors, and focus on the set *Reach* of correct processes that can reach all correct processes via exclusively eventually timely links and other correct processes. They show that $\diamond\mathcal{P}$ cannot be implemented if *Reach* does not contain all the correct processes. Similarly, they show that $\diamond\mathcal{S}$ (and hence Ω) cannot be implemented if *Reach* does not contain at least one correct process. In both cases, the subgraph formed by correct processes and eventually timely links in their system model must contain at least c arcs (e.g., in a ring topology), with $c \leq n - 1$ if at least one process crashes. In terms of our system model, these arcs correspond to our c links carrying messages forever.

5.5 Communication-Optimal Implementations of $\diamond\mathcal{P}$

In this section we introduce two approaches to the design of communication-optimal failure detection algorithms implementing $\diamond\mathcal{P}$. The approaches differ in how failure suspicions are managed. The first one uses an eager strategy in order to get low detection latencies. The second one is much more conservative in order to have a low sporadic communication overhead.

The first approach is based on every process consistently managing a local balance of suspicions and refutations for any other process. When a process p suspects another process q, p broadcasts a suspicion to every process, including q. If q has not failed, upon reception of that suspicion it will broadcast in the same way a refutation. Suspicions increment the corresponding balance, while refutations decrement it. With this strategy, eventually every correct process will permanently have a positive balance for every incorrect process, and a zero balance for every correct process. Observe that a reliable broadcast of suspicions and refutations is required in order to have consistent balances.

The alternative approach to the global spread of suspicions and refutations is to let a process to manage only suspicions in its *neighborhood* in the ring. In this approach, a process p will be in charge of the detection of incorrect processes between p's correct predecessor in the ring and p. The ring arrangement is used to propagate information about failures, piggybacked on periodic heartbeat messages, to all processes in a lazy way.

5.5.1 An Algorithm Using Reliable Broadcast

In this section, we propose a first communication-optimal implementation of $\diamond\mathcal{P}$ that uses Reliable Broadcast (see appendix B). In the algorithm, each process sends heartbeats to its successor in the ring, and monitors its predecessor by waiting heartbeats from it. Algorithm 5.1 presents the algorithm in detail, which uses a $Balance_p$ variable for every process p, accounting suspicions and refutations for every process.

Algorithm 5.1: Communication-optimal $\diamond\mathcal{P}$ using Reliable Broadcast

{Every process p executes the following}

1 **Procedure** *main()*
2 $pred_p \leftarrow pred(p)$ {p's estimation of its nearest correct predecessor in the ring}
3 $succ_p \leftarrow succ(p)$ {p's estimation of its nearest correct successor in the ring}
4 **foreach** $q \in \Pi$ **do**
5 $\Delta_p(q) \leftarrow$ default time-out interval {$\Delta_p(q)$ denotes the duration of p's time-out for q}
6 $Balance_p(q) \leftarrow 0$

7 || Task 1: **repeat periodically** {Sending heartbeats}
8 **if** $succ_p \neq p$ **then** send (ALIVE, p) to $succ_p$

9 || Task 2: **repeat periodically** {Checking time-outs}
10 **if** ($pred_p \neq p$ **and** p did not receive (ALIVE, $pred_p$) during the last $\Delta_p(pred_p)$ ticks of p's clock) **then** {time-out}
11 R-broadcast (SUSPICION, p, $pred_p$)

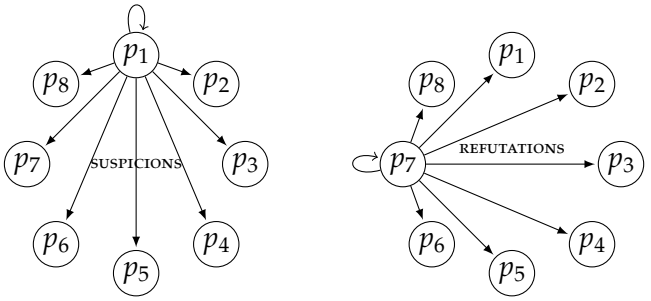
12 || Task 3: **w**hen r-deliver (SUSPICION, q, r) {Processing SUSPICIONS}
13 $Balance_p(r) \leftarrow Balance_p(r) + 1$
14 *update_pred_and_succ*()
15 **if** $r = p$ **then** R-broadcast (REFUTATION, p)

16 || Task 4: **w**hen r-deliver (REFUTATION, q) {Processing REFUTATIONS}
17 $Balance_p(q) \leftarrow Balance_p(q) - 1$
18 $\Delta_p(q) \leftarrow \Delta_p(q) + 1$ {not needed if $q = p$}
19 *update_pred_and_succ*()

20 **Procedure** *udpate_pred_and_succ()*
21 **if** $\forall r : Balance_p(r) > 0$ **then**
22 $pred_p \leftarrow p$
23 $succ_p \leftarrow p$
24 **else**
25 $pred_p \leftarrow p$'s nearest predecessor r in the ring such that $Balance_p(r) \leq 0$
26 $succ_p \leftarrow p$'s nearest successor r in the ring such that $Balance_p(r) \leq 0$

If $Balance_p(q) > 0$ with $q \neq p$, then p suspects q; else, q is trusted by p. As we will see, $Balance_p$ provides the properties of $\diamond\mathcal{P}$. Every process p starts sending periodically an (ALIVE, p) message to its successor in the ring, denoted by the variable $succ_p$ (Task 1). Also, every process p waits for periodical (ALIVE, $pred_p$) messages from its predecessor in the ring, denoted by the variable $pred_p$. If p does not receive such a message on a specific time-out interval of $\Delta_p(pred_p)$, then p suspects that $pred_p$ has crashed, and R-broadcasts a (SUSPICION, p, $pred_p$) message (Task 2), as shown in Figure 5.2a[3] (p_1

[3]Note that relayed messages caused by Reliable Broadcast are not shown.



(a) p_1 suspects p_7 **(b)** p_7 refutes p_1's suspicion

Figure 5.2: Sporadic communication in the communication-optimal Algorithm 5.1

suspects p_7). In Task 3, when p R-delivers a (SUSPICION, q, r) message, p increments $Balance_p(r)$ and calls the *update_pred_and_succ*() procedure. Besides this, if $r = p$, i.e., p has been erroneously suspected by q, p R-broadcasts a (REFUTATION, p) message (Figure 5.2b).

In Task 4, when p R-delivers a (REFUTATION, q) message, p decrements $Balance_p(q)$, increments $\Delta_p(q)$, in order to avoid premature suspicions in the future, and calls the *update_pred_and_succ*() procedure. Variables $pred_p$ and $succ_p$ are updated from $Balance_p$ to the nearest predecessor and the nearest successor in the ring having a non-positive balance respectively.[4] If all the components of the $Balance_p$ vector are positive, then p sets both $pred_p$ and $succ_p$ to p.

Correctness Proof

We show now that Algorithm 5.1 implements a failure detector of class $\diamond\mathcal{P}$ and that it is communication-optimal. In the proof, we consider that all the time instants are after all the incorrect processes have already crashed, and all the messages they have sent/R-broadcast before crashing have already been received/R-delivered. We start making the following observations.

Observation 5.1: *$\forall p \in C$, eventually and permanently $Balance_p(p) = 0$. This derives from the following: (1) initially, $Balance_p(p) = 0$, (2) for every message (SUSPICION, $-$, p) that p R-delivers in Task 3, eventually p R-delivers in Task 4 a (REFUTATION, p) message that compensates the previous increment of $Balance_p(p)$, and (3) eventually every correct process stabilizes with its correct predecessor in the ring, after which it stops R-broadcasting suspicions.*

Observation 5.2: *$\forall p$, if $pred_p = q$ with $q \neq p$ then $Balance_p(q) \leq 0$. Also, if $succ_p = r$ with $r \neq p$ then $Balance_p(r) \leq 0$. This derives directly from the fact that both $pred_p$ and $succ_p$ are only updated by p inside the procedure update_pred_and_succ().*

[4]Here we informally use the terms *nearest predecessor* (or *nearest successor*) of a process p to denote the first process preceding (or succeeding) p following the ring arrangement and fitting a particular condition.

Observation 5.3: *For every pair of correct processes p, q, when no more failure suspicions occur, $Balance_p(r) = Balance_q(r)$ for every process r. By the properties of Reliable Broadcast, both p and q R-deliver the same set of (*SUSPICION*, $-$, r) and (*REFUTATION*, r) messages. Consequently, both p and q apply the same modifications to $Balance_p(r)$ and $Balance_q(r)$ respectively.*

Lemma 5.1: *For every pair of consecutive correct processes q, p in the ring, eventually p stops R-broadcasting (*SUSPICION*, p, q) messages.*

Proof. The proof is by contradiction. Assume that p R-broadcasts suspicion messages, (SUSPICION, p, q), infinitely often. Since both p and q are correct, for each message (SUSPICION, p, q) q will R-broadcast a refutation message, (REFUTATION, q), that will be R-delivered by p. Upon R-delivery, p will increment $\Delta_p(q)$. Since the communication link between q and p is eventually timely, eventually $\Delta_p(q)$ will reach the unknown bound on message transmission times, after which p will receive an (ALIVE, q) message always before $\Delta_p(q)$ expires, and p will no more suspect q in Task 2. This contradicts the fact that p suspects q infinitely often. □

Lemma 5.2: *For every pair of non-consecutive correct processes q, p in the ring, eventually p stops R-broadcasting (*SUSPICION*, p, q) messages.*

Proof. By Lemma 5.1, eventually p will permanently monitor another correct process r, being r its correct predecessor in the ring. After that, p will never R-broadcast any (SUSPICION, p, q) message any more. □

Lemma 5.3: *For every pair of correct processes p, q, eventually and permanently $Balance_p(q) = 0$.*

Proof. Follows from Lemma 5.1 and Lemma 5.2, and the fact that, being initially $Balance_p(q) = 0$, by the algorithm p R-delivers the same number of (SUSPICION, $-$, q) and (REFUTATION, q) messages, compensating the increment and decrement operations over $Balance_p(q)$. □

Lemma 5.4: *For every incorrect process q, eventually and permanently the value $Balance_p(q) > 0$ for every correct process p.*

Proof. Note that after q crashes it will not be able to R-broadcast any message (REFUTATION, q). Also, at least process r, being r the correct successor of q in the ring, will eventually R-broadcast a (SUSPICION, r, q) message that q will not refute, and consequently $Balance_r(q) > 0$ permanently. Then, by Observation 5.3, $Balance_p(q) > 0$ for every correct process p. □

Lemma 5.5: *Eventually, for every correct process p, $pred_p$ will be permanently set to p's correct predecessor in the ring, and $succ_p$ will be permanently set to p's correct successor in the ring.*

Proof. There are two cases to consider. If $c = 1$, then by definition both p's correct predecessor in the ring and p's correct successor in the ring are p itself, and the lemma follows directly from Lemma 5.4, Observation 5.1 and the procedure *update_pred_and_succ*(). Otherwise, i.e., if $c > 1$, the lemma follows directly from Lemma 5.3, Lemma 5.4, Observation 5.2 and the procedure *update_pred_and_succ*(). □

Theorem 5.3: *Algorithm 5.1 implements a failure detector of class $\diamond\mathcal{P}$.*

Proof. From Lemma 5.3 and Lemma 5.4, for every correct process p, eventually and permanently $Balance_p(q) = 0$ for every $q \in C$, and $Balance_p(r) > 0$ for every $r \in F$. The rule "if $Balance_p(q) > 0$, then p suspects q; else, p does not suspect q" provides the properties of strong completeness and eventual strong accuracy of $\diamond\mathcal{P}$. □

Theorem 5.4: *Algorithm 5.1 is communication-optimal, i.e., eventually only* c *links carry messages forever.*

Proof. From Lemma 5.5, for every correct process p, eventually and permanently $succ_p$ will be set to p's correct successor in the ring and, by Task 1, p will send (ALIVE, p) messages to it forever. No other periodical messages will be sent. Furthermore, since no more suspicions will occur, no new SUSPICION (and hence REFUTATION) messages will be broadcast. Thus, if there are c correct processes in the system, just a number of c unidirectional links will be permanently used. □

Observe that if there is just one correct process in the system, i.e., $c = 1$, Algorithm 5.1 eventually uses no links, by an optimization introduced in Task 1. Hence, when $c = 1$ both $\diamond\mathcal{P}$ and Ω can be implemented using 0 links carrying messages forever.

Finally, although we have initially assumed in the system model that all the communication links were reliable and eventually timely, in a given execution of Algorithm 5.1 it is sufficient that this behaviour applies only to the c links that eventually form the ring of correct processes, i.e., the links from every correct process to its correct successor in the ring. The rest of links can be asynchronous and/or lossy.

5.5.2 An Algorithm Using One-to-One Local Communication

In this section, we present a communication-optimal $\diamond\mathcal{P}$ algorithm that uses only one-to-one local communication to manage suspicions. In the algorithm, a process p will be in charge of the detection of incorrect processes between p's correct predecessor in the ring and p. Assuming that simple heartbeat messages circulate around the ring, this strategy only gives *weak* completeness and eventual strong accuracy, and hence the resulting failure detector will be of the class $\diamond\mathcal{Q}$. Henceforth, a further transformation is needed to get a failure detector of the class $\diamond\mathcal{P}$. In this way, the approach we follow to design the algorithm is incremental: first we present the algorithm implementing $\diamond\mathcal{Q}$ and prove its correctness, and next we give a simple transformation into $\diamond\mathcal{P}$ which preserves communication optimality and low sporadic communication overhead.

Algorithm 5.2: Communication-optimal $\diamond\mathcal{Q}$ using local one-to-one communication

{Every process p executes the following}

Procedure *main()*
  $pred_p \leftarrow pred(p)$ {p's estimation of its nearest correct predecessor in the ring}
  $succ_p \leftarrow succ(p)$ {p's estimation of its nearest correct successor in the ring}
  foreach $q \in \Pi$ **do** $\Delta_p(q) \leftarrow$ default time-out interval {$\Delta_p(q)$ denotes the duration of p's time-out for q}
  $L_p \leftarrow \emptyset$ {L_p provides the properties of $\diamond\mathcal{Q}$ }

|| Task 1: **repeat periodically** {Sending heartbeats}
  if $succ_p \neq p$ **then** send (ALIVE, p) to $succ_p$

|| Task 2: **repeat periodically** {Checking time-outs}
  if ($pred_p \neq p$ **and** p did not receive (ALIVE, $pred_p$) during the last $\Delta_p(pred_p)$ ticks of p's clock) **then** {time-out}
    $L_p \leftarrow L_p \cup \{pred_p\}$ {p suspects $pred_p$ has crashed}
    send (SUSPICION, p) to $pred_p$
    update_pred_and_succ()

|| Task 3: when receive (SUSPICION, q) from some q {Processing SUSPICIONS}
  $L_p \leftarrow L_p \cup \{p, \ldots, q\} - \{p, q\}$
  update_pred_and_succ()
  foreach $r \in \{p, \ldots, q\} - \{p, q\}$ **do** send (PROBE, p) to r
  send (ALIVE, p) to q

|| Task 4: when receive (ALIVE, q) from some q {Processing ALIVES}
  if $q \in L_p$ **then**
    $L_p \leftarrow L_p - \{q\}$
    update_pred_and_succ()
    $\Delta_p(q) \leftarrow \Delta_p(q) + 1$

|| Task 5: when receive (PROBE, q) from some q {Processing PROBES}
  send (ALIVE, p) to q

Procedure *udpate_pred_and_succ()*
  $pred_p \leftarrow p$'s nearest predecessor r in the ring such that $r \notin L_p$
  $succ_p \leftarrow p$'s nearest successor r in the ring such that $r \notin L_p$
  if $pred_p \neq p$ **then** $L_p \leftarrow \{pred_p, \ldots, succ_p\} - \{pred_p, p, succ_p\}$

Implementing $\diamond\mathcal{Q}$

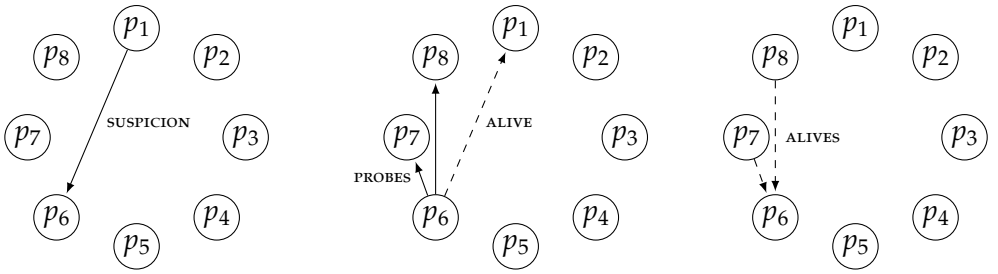
Algorithm 5.2 presents a communication-optimal implementation of $\diamond\mathcal{Q}$. Every process p has a local set of suspected processes, L_p, and two variables, $pred_p$ and $succ_p$, denoting respectively the process that p is monitoring and the process to which p is periodically sending heartbeat messages (ALIVE, p) by Task 1. When a process p suspects by Task 2 the process it is monitoring, $pred_p$, p includes $pred_p$ in L_p, sends a suspicion message (SUSPICION, p) to $pred_p$ (in Figure 5.3a, p_1 suspects p_6), and updates $pred_p$ (and $succ_p$ if required) accordingly. Observe that Task 2 does not include any explicit mechanism for p to tell its new predecessor to start sending heartbeats to it. The new predecessor of p will have to be suspected once by p in order to set its successor to p by Task 3, as we will explain next.

If a suspected process p is correct or has not crashed yet, when it receives (SUSPICION, q) by Task 3, p will suspect every process from p to q (both excluded), since all of them

have been also suspected by q. Consequently, process q becomes the new successor of p, and hence, if p does not crash, q will receive periodical (ALIVE, p) messages from p, as shown in Figure 5.3b. Furthermore, a sporadic (PROBE, p) message is sent by p to every process r from p to q (both excluded), in order to know if r has actually crashed or not (also shown in Figure 5.3b, in which p_6 probes p_7 and p_8). When a process p receives a (PROBE, q) message, it just sends an (ALIVE, p) message to q, in order to give q the opportunity to set $succ_q$ (and exceptionally $pred_q$) to p (in Figure 5.3c, $p7$ and $p8$ notify p_6 that they are alive).

On the reception of an (ALIVE, q) message coming from a process $q \in L_p$, by Task 4 a process p will remove q from L_p, update $pred_p$ (and $succ_p$ if required), and increment the time-out interval with respect to q, $\Delta_p(q)$.

The PROBE messages used in this algorithm avoid a process to send periodically ALIVE messages to crashed processes, a scenario that can happen in communication-efficient algorithms (see Figure 5.1, where processes p_7 and p_8 have crashed). Hence, the proposed probing mechanism is key to get communication optimality.



(a) p_1 suspects p_6

(b) p_6 probes p_7 and p_8, and sends ALIVE to p_1

(c) p_7 and p_8 reply to PROBE with ALIVE

Figure 5.3: Sporadic communication in the communication-optimal Algorithm 5.2

Correctness Proof

We show now that Algorithm 5.2 implements a failure detector of class $\diamond\mathcal{Q}$ and that it is communication-optimal. Given any process p, we denote by $corr\_pred_p$ the correct predecessor of p in the ring. Similarly, we denote by $corr\_succ_p$ the correct successor of p in the ring. The key of the proof is to show that eventually and permanently $pred_p = corr\_pred_p$ and $succ_p = corr\_succ_p$ for every correct process p. In other words, the ring stabilizes in terms of both the *pred* and *succ* variables of processes, which guarantees the correct construction of the sets L_p of suspected processes.

We assume that every task is executed as a critical section. We start by making the following observations:

Observation 5.4: *$p \notin L_p$ permanently for every process p.*

Observation 5.5: *$L_p = \{pred_p, \ldots, succ_p\} - \{pred_p, p, succ_p\}$ permanently after the execution of any task for every process p.*

Observe that, whenever L_p is modified by Task 2, Task 3 or Task 4 of p, $pred_p$ and $succ_p$ (and L_p itself) are updated by the *update_pred_and_succ()* procedure accordingly.

Observation 5.6: *Whenever a process q is included by a correct process p in L_p, p will send a message (of type* SUSPICION *in Task 2 or type* PROBE *in Task 3) to q, and, if q is correct, p will eventually receive an (ALIVE, q) message sent by Task 3 or Task 5 of q.*

For the rest of the proof we will assume that any time instant t considered is larger than a time t_{base} that occurs after the stabilization time GST (i.e., $t_{base} > GST$), after every incorrect process has crashed, and after all messages sent by incorrect processes have been received. Note that this eventually happens. Hence, any new message received has necessarily been sent by a correct process.

Lemma 5.6: *For every correct process p, eventually and permanently $pred_p = corr\_pred_p$.*

Proof. For a correct process p, let $pred_p = r$ such that $r \neq corr\_pred_p$. For a given time in the execution of the algorithm, one of the following cases applies:

Case 1: assume first that $r \in \{corr\_pred_p, \ldots, p\} - \{corr\_pred_p, p\}$. Observe that, with this assumption, r is by definition not correct, as well as any other process in that range. Therefore, r will be eventually included in L_p (by Task 2 or Task 3 of p), and, since r has already crashed, it will not be able to send an (ALIVE, r) message to p, remaining in L_p forever.

Case 2: assume now that $r \notin \{corr\_pred_p, \ldots, pred(p)\}$. By Observation 5.5, $corr\_pred_p \in L_p$. Observe that $corr\_pred_p$ could have been included in L_p by Task 2 or by Task 3 of p. Since $corr\_pred_p$ is by definition correct, by Observation 5.6 eventually p will receive and $(\text{ALIVE}, corr\_pred_p)$ message. At this point, $pred_p$ will be set to $corr\_pred_p$.

Since every time $corr\_pred_p$ has been suspected by p, $\Delta_p(corr\_pred_p)$ is incremented by Task 4 of p, and since the communication link between $corr\_pred_p$ and p is eventually timely, eventually $\Delta_p(corr\_pred_p)$ will reach the unknown bound on message transmission times, after which p will receive the periodical $(\text{ALIVE}, corr\_pred_p)$ messages always before timer $\Delta_p(corr\_pred_p)$ expires and $corr\_pred_p$ will not be suspected by p anymore. □

Lemma 5.7: *For every correct process p, eventually and permanently $succ_p = corr\_succ_p$.*

Proof. The proof is by contradiction. By Lemma 5.6, eventually and permanently $pred_{corr\_succ_p} = p$. Assume that $succ_p \neq corr\_succ_p$. Since p is not sending by Task 1 periodical (ALIVE, p) messages to $corr\_succ_p$, then by Task 2 $corr\_succ_p$ will eventually suspect p and modify $pred_{corr\_succ_p}$, which contradicts Lemma 5.6. □

Theorem 5.5: *Algorithm 5.2 implements a failure detector of class $\diamond\mathcal{Q}$.*

Proof. From Lemmas 5.6 and 5.7, for every correct process p, eventually p will be in the stable ring formed by correct processes. Otherwise, p is incorrect, and by Lemma 5.6 eventually and permanently $pred_{corr\_succ_p} = corr\_pred_p$. By Observation 5.5, p will eventually and permanently be included in $L_{corr\_succ_p}$. As a consequence, eventually and permanently p will be included in the set of suspected processes of some correct process. This provides the weak completeness property of $\diamond\mathcal{Q}$.

By Observation 5.4, p is never included in L_p. Once the ring has stabilized, no correct process is included (in Task 2) in the set L_p of any correct process p. Hence, once the ring has stabilized, no correct process will be present in any set of suspected processes. This provides the eventual strong accuracy property of $\diamond\mathcal{Q}$. □

Theorem 5.6: *Algorithm 5.2 is communication-optimal, i.e., eventually only* c *links carry messages forever.*

Proof. From Lemma 5.7, for every correct process p, eventually and permanently $succ_p$ will be set to p's correct successor in the ring and, by Task 1, p will send (ALIVE, p) messages to it forever. No other periodical messages will be sent. Furthermore, since no more suspicions will occur, no new SUSPICION (and hence sporadic PROBE or ALIVE) messages will be sent. Thus, if there are c correct processes in the system, just a number of c unidirectional links will be permanently used. □

A similar reasoning as the one made for the previous algorithm can be made for Algorithm 5.2 regarding the minimal assumptions on communication reliability and synchrony required by the algorithm. In this case, in a given execution of Algorithm 5.2 it is sufficient that the links from every correct process to both its correct successor and predecessor in the ring are reliable and eventually timely, i.e., $2c$ links. The rest of links can be asynchronous and/or lossy.

Transforming $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$

In this section we present a transformation of the $\diamond\mathcal{Q}$ algorithm of the previous section into $\diamond\mathcal{P}$. The transformation preserves the communication optimality and low sporadic communication overhead of Algorithm 5.2.

Algorithm 5.3, which implements $\diamond\mathcal{P}$, is obtained from Algorithm 5.2 by adding a set G_p to every process p. G_p is included into the ALIVE messages sent by p. In the algorithm, additions and removals of processes to L_p are also applied to G_p. To build G_p from the set G_{pred_p} received in Task 4, process p adds the processes between $pred_p$ and p to G_{pred_p} (both $pred_p$ and p excluded). In other words, p relies on its predecessor for suspicions beyond its directly monitored neighborhood.

Correctness Proof

We show now that Algorithm 5.3 implements a failure detector of class $\diamond\mathcal{P}$ and that it is also communication-optimal. The key of the proof is to show that Algorithm 5.3

Algorithm 5.3: Communication-optimal $\diamond\mathcal{P}$ using local one-to-one communication

```
{Every process p executes the following}
1  Procedure main()
2    pred_p ← pred(p)                          {p's estimation of its nearest correct predecessor in the ring}
3    succ_p ← succ(p)                          {p's estimation of its nearest correct successor in the ring}
4    foreach q ∈ Π do Δ_p(q) ← default time-out interval   {Δ_p(q) denotes the duration of p's time-out for q}
5    L_p ← ∅ ; G_p ← ∅                         {L_p provides the properties of ◇Q; G_p provides the properties of ◇P }

6  || Task 1: repeat periodically                                                    {Sending heartbeats}
7    if succ_p ≠ p then send (ALIVE, p, G_p) to succ_p

8  || Task 2: repeat periodically                                                    {Checking time-outs}
9    if ( pred_p ≠ p and p did not receive (ALIVE, pred_p, −)
          during the last Δ_p(pred_p) ticks of p's clock ) then                     {time-out}
10     L_p ← L_p ∪ {pred_p}; G_p ← G_p ∪ {pred_p}                                   {p suspects pred_p has crashed}
11     send (SUSPICION, p) to pred_p
12     update_pred_and_succ()

13 || Task 3: when receive (SUSPICION, q) from some q                              {Processing SUSPICIONS}
14   L_p ← L_p ∪ {p,...,q} − {p,q}; G_p ← G_p ∪ {p,...,q} − {p,q}
15   update_pred_and_succ()
16   foreach r ∈ {p,...,q} − {p,q} do send (PROBE, p) to r
17   send (ALIVE, p, G_p) to q

18 || Task 4: when receive (ALIVE, q, G_q) from some q                             {Processing ALIVES}
19   if q ∈ L_p then
20     L_p ← L_p − {q}
21     update_pred_and_succ()
22     Δ_p(q) ← Δ_p(q) + 1
23   if q = pred_p then G_p ← (G_q ∪ {pred_p,...,p}) − {pred_p, p}

24 || Task 5: when receive (PROBE, q) from some q                                  {Processing PROBES}
25   send (ALIVE, p, G_p) to q

26 Procedure udpate_pred_and_succ()
27   pred_p ← p's nearest predecessor r in the ring such that r ∉ L_p
28   succ_p ← p's nearest successor r in the ring such that r ∉ L_p
29   if pred_p ≠ p then L_p ← {pred_p,...,succ_p} − {pred_p, p, succ_p}
```

is a transformation of Algorithm 5.2 into $\diamond\mathcal{P}$ preserving communication optimality. Observe that in Task 2 and Task 3 exactly the same operations are applied to L_p and to G_p. Finally, in Task 4, the global set of suspected processes G_p is built by adding to G_{pred_p} the processes between $pred_p$ and p (both $pred_p$ and p excluded).

Observation 5.7: *$p \notin G_p$ permanently for every process p.*

Theorem 5.7: *Algorithm 5.3 implements a failure detector of class $\diamond\mathcal{P}$.*

Proof. Observe that, by the construction of the set G_p in Tasks 2, 3, and 4, G_p is equivalent to L_p in the range $\{pred_p, \ldots, p\}$. By Lemmas 5.6 and 5.7, and by Task 4 of p, eventually G_p will include every incorrect process, providing the strong completeness property of $\diamond\mathcal{P}$. By Observation 5.7 and again by Lemmas 5.6 and 5.7 and Task 4 of p,

no correct process will be eventually included in G_p, preserving the eventual strong accuracy property of $\diamond\mathcal{P}$. □

Theorem 5.8: *Algorithm 5.3 is communication-optimal, i.e., eventually only* c *links carry messages forever.*

Proof. Follows directly from the fact that no additional message is sent in Algorithm 5.3 with respect to Algorithm 5.2. □

5.5.3 Performance Analysis

Obviously, an algorithm implementing a failure detector of a given class must satisfy the properties defined for that class. When implementing a failure detector, besides satisfying the properties of the class it belongs to, performance should be also taken into account. In this work, we focus on the following two performance issues:

i) Scalability, to allow a failure detector to be deployed in networks with a high number of nodes and/or heterogeneous links. A scalable failure detection algorithm should use a low number of links and avoid all-to-all communication patterns.

ii) Quality-of-service, involving several parameters, such as detection latency or stabilization time *[Chen, Toueg, and Aguilera, 2002]*, which affect the responsiveness of the system. For example, when a crashed process q is suspected by a process p, every correct process should be informed as soon as possible in order to provide low detection latencies. In this regard, a one-to-all communication pattern for suspicion propagation can be helpful. However, when the suspicion is erroneous such a communication pattern could become a drawback, since it propagates the erroneous suspicion in the system.

Keeping in mind scalability and quality-of-service, we now consider a set of interesting properties for an algorithm that implements a failure detector:

Communication efficiency. Recall from Section 5.3 that in a communication-efficient algorithm only n unidirectional links carry messages forever, being n the number of processes in the system. Observe that communication efficiency refers to a *permanent* behaviour that will hold eventually. In large systems, communication-efficient algorithms will be more scalable than non communication-efficient algorithms.

Low sporadic overhead. Besides the permanent communication cost due to periodic messages (heartbeats), which is addressed by the communication efficiency property, an algorithm implementing a failure detector can involve *sporadic* extra messages caused by failure suspicions, resulting in a peak overhead. There is a

trade-off in the way this sporadic traffic is managed. On the one hand, a one-to-one communication pattern can be used (e.g., by using a logical ring arrangement); this way communication overhead is reduced and, henceforth, it provides better scalability. On the other hand, one-to-all (or even all-to-all) communication leads to provide low detection latencies.

Communication locality. This property is two-folded. On the one hand, we will say that a failure detection algorithm has *periodic communication locality* when periodic, permanent monitoring messages (i.e., heartbeats), are sent to some process(es) in the neighborhood of the sender process. Observe that a logical ring arrangement of processes based on proximity naturally provides this property. Similarly, we will say that a failure detection algorithm has *sporadic communication locality* when messages sent as a consequence of a suspicion are sent likely to some process(es) in the neighborhood of the suspecting process. Again, trade-offs should be considered between the good scalability associated to communication locality, and the potential good quality-of-service provided by the use of one-to-all and all-to-all communication patterns.

In the light of these properties, in this section we analyze the performance of the communication-optimal $\diamond\mathcal{P}$ algorithms. We also include in the analysis the algorithm of *[Chandra and Toueg, 1996]* as a reference, as well as the simplest of the communication-efficient algorithms in *[Larrea et al., 2007b]* These two algorithms can be found in Appendix C. The goal is to highlight the strengths and weakness of each approach and set the trade-offs to be considered in the implementation of failure detectors for particular scenarios. Specifically, we focus on the following performance parameters:

- Communication efficiency/optimality. Number of unidirectional links that carry messages forever.
- Sporadic communication overhead. Number of extra messages exchanged to manage an erroneous suspicion.
- Detection latency (or system responsiveness). Upon a crash, time elapsed until every alive process permanently suspects the crashed process.
- Communication locality. Scope of the messages. An algorithm exhibits (periodic or sporadic) communication locality when processes send (periodic or sporadic) messages mainly to their neighborhood.

Table 5.1 summarizes the performance of the algorithms. Note that sporadic communication locality is not applicable to the algorithms in *[Chandra and Toueg, 1996; Larrea et al., 2007b]*, since they do not have sporadic communication at all (sporadic overhead is null). Observe also that Algorithm 5.3, besides being communication-optimal, has a low sporadic overhead due to its one-to-one communication pattern, and has both periodic and sporadic communication locality. This is at the cost of a linear detection

latency, in contrast to the uniform detection latency of the algorithms using a one-to-all communication pattern.

Algorithm	Comm. efficiency (# links used forever)	Sporadic overhead (# extra messages)	Detection latency	Comm. locality	
				Periodic	Sporadic
Chandra-Toueg *[Chandra and Toueg, 1996]*	$c(n-1)$	0	T_h	No	—
Comm.-efficient *[Larrea et al., 2007b]*	n	0	$c\ T_h$	Yes	—
Comm.-optimal Alg. 5.1	c	$2n^2$	T_h	Yes	No
Comm.-optimal Alg. 5.3	c	2ℓ	$c\ T_h$	Yes	Yes

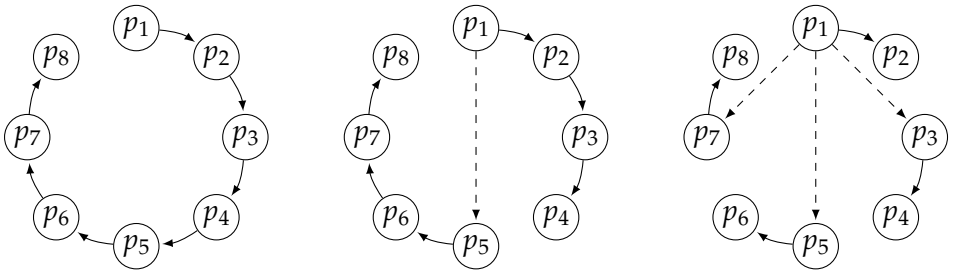
Table 5.1: Performance analysis of $\diamond\mathcal{P}$ algorithms. Sporadic overhead and detection latency figures are approximate (ℓ denotes the cardinality of the local set of suspected processes, usually $\ell \ll n$)

In Table 5.1, detection latency is estimated on the basis of the term T_h, which denotes the mean delay for sending a new, periodic, heartbeat message, by which the information about suspected processes is relayed to the successor process in the ring in the algorithm in *[Larrea et al., 2007b]* and in Algorithm 5.3. Assuming that the transmission delay of any message is negligible with respect to T_h, the value of T_h can be estimated as one half of the periodicity of the task that sends periodic heartbeat messages (Task 1 in the algorithms proposed in this chapter).

The linear detection latency in Algorithm 5.3 is due to the fact that the information about a new suspicion has to circulate around the whole ring to reach every alive process. Nevertheless, a mechanism to speed-up the detection latency, at the price of losing sporadic communication locality, can be easily added. The mechanism consists in the suspecting process p directly sending the suspicion message not only to the suspected process but also to a subset of processes $\Lambda \subset \Pi$, i.e., adding shortcuts in the ring. In general, if $k = |\Lambda|$, while the communication overhead is incremented in at most $2k$ messages, assuming that shortcuts have been uniformly distributed, an estimation of the maximum detection latency results in $(\frac{n}{k+1})T_h$. In the limit, $\Lambda = \Pi - \{p\}$, and the detection latency is reduced to the minimum. The mechanism can be also applied to the communication-efficient algorithm in *[Larrea et al., 2007b]*.

Figure 5.4 shows an example of the shortcut mechanism to improve responsiveness in Algorithm 5.3. Without shortcuts, a failure suspicion propagates by heartbeats and reaches all alive processes in approximately $c\ T_h$ time. One or more shortcuts allow the suspicion to be propagated in parallel, reducing the detection latency. Of course, in case the suspicion is erroneous, such a mechanism makes the error to be propagated faster in the system. A way of avoiding this apparent drawback consists in adding a complementary shortcut mechanism for the refutation of a recent, erroneous, suspicion. Observe that neither of these suspicion/refutation shortcut mechanisms compromises

the optimality of Algorithm 5.3, since eventually, i.e., when the ring formed by correct processes stabilizes, they are not used any more.



(a) Without shortcuts **(b)** Adding one shortcut **(c)** Adding three shortcuts

Figure 5.4: Example of the shortcut mechanism to improve responsiveness

In general, a trade-off results between detection latency and communication overhead, being the latter either periodic or sporadic. It is worth noting that parameters related to communication overhead have been expressed in terms of the number of links and messages. Communication costs, however, strongly depend on network parameters as topology and link delays, specifically in wide area networks, resulting in not uniform message transmission delays. This fact can be exploited when using a ring arrangement as a pattern for communication. More precisely, communication costs associated to pairs of processes provide a hint for the initial ring configuration. Periodic heartbeat communication will take advantage of this. Furthermore, sporadic communication overhead, which is inherent to communication-optimal algorithms, results in practice less harmful for algorithms with this kind of local communication than for algorithms with global communication. Finally, as we have shown, the ability of using shortcuts in the formers provide an interesting basis to fit trade-offs between sporadic communication overhead and detection latency.

5.6 Conclusions

In this chapter, we have explored communication efficiency, a performance measure that refers to the number of unidirectional links that carry messages forever in an algorithm. We have shown that failure detector class $\diamond\mathcal{P}$ requires at least c unidirectional links to carry messages forever, being c the number of correct processes. Moreover, when at least one process crashes, c links are also required for $\diamond\mathcal{S}$ and Ω. We have proposed two ring-based communication-optimal $\diamond\mathcal{P}$ algorithms. Since these algorithms use exactly c unidirectional links to carry messages forever, it can be derived that communication optimality for $\diamond\mathcal{P}$ is achieved. Since $\diamond\mathcal{P}$ trivially implements Ω, communication optimality can be considered achieved also for $\diamond\mathcal{S}$ and Ω failure detectors.

One of the algorithms uses a Reliable Broadcast primitive to communicate suspicions and refutations, involving a quadratic number of messages. Since this can be a drawback in some scenarios, e.g., very large networks, we have proposed a second algorithm that uses exclusively one-to-one communication. This algorithm has some interesting properties which make it suitable for distributed applications deployed over wide area networks: (1) communication optimality, i.e., only c links carry messages forever, (2) low sporadic communication overhead to manage failure suspicions, i.e., the number of messages exchanged as a consequence of a suspicion is linear, and (3) communication locality, i.e., managing a failure suspicion at a process p only implies communicating with processes at p's neighborhood in the ring. This is of particular interest in networks where communication costs between pairs of processes are not uniform. If the logical ring of processes is arranged regarding communication cost criteria, both communication overhead and delays of periodic heartbeats will be minimized.

The second algorithm admits a flexible implementation. In its basic form, i.e., using exclusively local communication, it exhibits a linear detection latency. However, a mechanism based on shortcuts can reduce it. Moreover, this mechanism can be enabled or disabled without affecting the correctness of the algorithm. As a consequence, the mechanism can be applied partially and be precisely tuned in order to fit trade-offs between network overhead and QoS parameters.

Chapter 6

A Failure Detector for The General Omission Failure Model

The failure detector abstraction has been investigated in quite some detail in systems with merely crash failures, e.g., Chapter 5. Nevertheless, there is very little work on failure detection and consensus in message omission environments. In fact, it is not clear what a sensible definition of a failure detector (and consensus) is in such environments because the notion of a correct process can have several different meanings, e.g., a process with no failures whatsoever or a process which does not crash but just omits some messages.

We propose to focus on processes *connectedness* instead of *correctness*. Thus we present a new definition of connectedness in omission environments. Informally, a process is *in-connected* if it does not crash and, despite omissions, receives either directly or indirectly all messages that some correct process sends to it. Similarly, a process is *out-connected* if it does not crash and all messages it sends to some correct process are received by that correct process. From these definitions, every out-connected process can send information with no omissions to any in-connected process. Our goal is to include as many processes as possible in the set of processes that are able to participate "actively" in the consensus, i.e., out-connected processes can propose, while in-connected processes can decide.

Based on the previous definitions, we give a novel definition of a failure detector class in the omission model and we show how to implement it in a system with weak synchrony assumptions in the spirit of partial synchrony. The proposed failure detector class could be considered to be equivalent to $\diamond\mathcal{P}$ in terms of its completeness and accuracy properties, both adapted to the *connectedness* of the processes. We also give a novel definition of consensus in the general omission model by refining the termination property of consensus ("Every in-connected process eventually decides some value"), and an algorithm which uses the previously proposed failure detector class to solve consensus. The algorithm is an adaptation of the classic algorithm by *[Chandra and Toueg, 1996]* for the crash model.

Outline. In Section 6.1 we first present a related work to the failure detection in omission failure models. Section 6.2 addresses failure detection in the general omission failure model. In Section 6.3 we introduce the *in-connected* and *out-connected* concepts as a replacement to the classical *correct* concept. Then, in Section 6.4 we present the system model we will consider and the new failure detector class. The implementation of the failure detector is presented in Section 6.5. In addition, Section 6.6 redefines consensus based on connectedness and proposes an adaptation of the classical consensus algorithm of *[Chandra and Toueg, 1996]*. Section 6.7 shows how consensus and failure detector messages could be integrated. Finally, Section 6.8 points out the conclusions of the chapter.

6.1 Related work

Some other authors have previously investigated solving consensus in systems with omission failures. *[Dolev, Friedman, Keidar, and Malkhi, 1996 and 1997]* follow the failure detector approach to solve consensus, focusing on the $\diamond\mathcal{S}(om)$ failure detector class. *[Babaoglu, Davoli, and Montresor, 2001]* also follow the path of $\diamond\mathcal{S}$ to solve consensus in partitionable systems. *[Doudou, Garbinato, Guerraoui, and Schiper, 1999]* defined muteness failure detectors $\diamond M_{\mathcal{A}}$ to detect processes that stop sending top level application messages, e.g., consensus messages. More recently, *[Delporte-Gallet, Fauconnier, and Freiling, 2005; Delporte-Gallet, Fauconnier, Freiling, Penso, and Tielmann, 2007; Cortiñas et al., 2007]* have proposed several consensus algorithms based on failure detectors of the classes Ω and $\diamond\mathcal{P}$.

The failure detector algorithms of *[Delporte-Gallet et al., 2005 and 2007]* rely on piggybacking information of previous messages in order to cope with transient omissions. The system model proposed in *[Cortiñas et al., 2007]* allows processes to communicate indirectly so that more processes can participate actively in the consensus protocol.

6.2 Failure Detection in the General Omission Failure Model

The general omission failure model includes both send and receive omissions, which can be transient. When considering this failure model, an omission happens when a message m is omitted at sending (send omission) or at receiving (receive omission). Observe that, when there is an omission of a message m sent by p to q, both communicating processes can be involved in the failure; if there has been a send omission, then the blame should be put on p. However, if there has been a receive omission, q is a faulty process.

Impossibility Result for deterministically distinguish correct and faulty processes. We give here a proof sketch that the classical $\diamond\mathcal{P}$ is impossible to implement in the general omission model. The proof is by contradiction. Assume that we have an algorithm A that implements $\diamond\mathcal{P}$ in the general omission model. Consider a run R_1 in which just one message m sent by a process p to a correct process q is not received by q,

because p has suffered a send omission, i.e., p is incorrect in R_1, which is represented in Figure 6.1a. Consider another run R_2 in which p is correct and successfully sends m to q, but q does not receive m because it has suffered a receive omission (because q is an incorrect process). Figure 6.1b illustrates this second case. No other omission failure occurs in R_1 or R_2. For any correct process r, if r does not eventually and

(a) p suffers a send omission **(b)** q suffers a receive omission

Figure 6.1: Undistinguishable failures: (a) send-omission and (b) receive-omission

permanently suspect p in $R1$, then the strong completeness property of $\diamond\mathcal{P}$ is violated. Hence, r will eventually and permanently suspect p in R_1. Since both runs R_1 and R_2 are indistinguishable for r, in run R_2 r will also eventually and permanently suspect p, therefore violating the eventual strong accuracy property of $\diamond\mathcal{P}$. Thus, we can conclude that it is impossible to fulfil the classical completeness and accuracy properties of the $\diamond\mathcal{P}$ failure detector.

Remark 6.1: *In the general omission model there are undetectable failures, which make impossible to reliable detect correct processes.*

To set an example, if there is at least one crashed process and there is an unknown number of processes which can potentially omit messages, some processes could suffer send omissions with the crashed process and those omissions would be undetectable.

From correct processes to *good* processes. Since it is impossible to reliably detect correct processes, we propose to consider *good* processes instead. The *good processes* approach has already been followed by previous works, e.g., *[Guerraoui et al., 1999; Aguilera, Chen, and Toueg, 2000]*, defining processes *goodness* depending on the considered failure model. In the general omission failure model we consider as *good* processes those processes that are able to compute (not crashed) and to communicate without omissions with a majority of processes in the system. Hence, we define *goodness* of processes based on their connectivity.[1]

This approach allows some faulty processes to be considered as *good* processes, as long as they meet the requirements to be considered *good*. From this observation, we can state that some assumptions related to the correctness of processes can be replace by assumptions related to the *goodness* of processes. To set an example, instead of assuming a majority of correct processes, we could assume a majority of *good* processes.

In this chapter we will distinguish *in-connected* processes and *out-connected* processes. Additionally, in Chapter 8 we simplify this classification by only considering *well connected* processes.

[1]Observe that a crashed process is a process which stops sending and receiving processes, i.e., it has null connectivity.

6.3 From Correct Processes to *In/Out-Connected* Processes

In order to tolerate some transient omissions, we admit the possibility of indirect communication between two processes using a relaying mechanism. For example, if there are omissions in the communication channel from a process p to another process q, but both of them have no omissions with a third process r, process p could indirectly communicate with q through r without any omission. This way, a process will be considered a *good* or *connected* process as long as it is able to communicate with a correct process — and hence with any correct process — without any omission, even if it has suffered some omissions with many other processes.

Furthermore, we should notice that the *connectedness* of a process can be asymmetric, since it can suffer send omissions and receive omissions independently, e.g., a process can be able to send to any correct process, but not be able to receive from any of them (because it has had too many receive omissions). Following this motivation, we consider the following classes of processes, based on their ability to communicate (we give only rough and intuitive definitions here and fully formalize these notions later in Sect. 6.4):

- A process is *in-connected* if it does not crash and it receives all messages that some correct process sends to it.
- A process is *out-connected* if it does not crash and all messages it sends to some correct process are received by that correct process.

Based on these definitions, correct processes are both in-connected and out-connected. Observe that every out-connected process can send information to any in-connected process with no omissions. Figure 6.2 shows an example where arcs represent links with no omissions (they are not shown for the set of correct processes). Processes p and q are out-connected, while process s is in-connected, and processes r and v are both in-connected and out-connected. Finally, process u is neither in-connected nor out-connected.

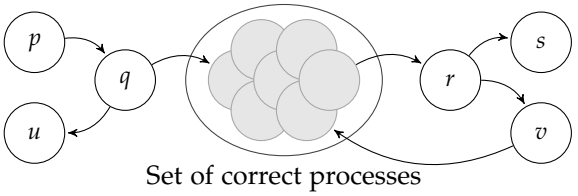


Figure 6.2: Examples for classes of processes

6.3.1 *In/Out-Connected* Processes and Consensus

[Dwork et al., 1988] showed that consensus can be solved in partially synchronous systems with crashes or omissions provided a majority of correct processes.

Nevertheless, even though we assume a majority of correct processes, we can also allow some *faulty* processes to participate in the consensus provided that they keep their ability to compute (no crash) and to communicate without omissions with at least one correct process (in- and out-connectivity). In fact, the assumption of a majority of correct processes is not strictly necessary. Actually, it is sufficient to have a majority of processes that are both in- and out-connected, as we will show in this chapter.

6.4 System Model

Following the system model presented in Chapter 2, we model an algorithm as a set of deterministic automata (one per process). Events are generated at execution or sending/reception of messages. Those events are univocally identified by a fictional global clock and processed following a FIFO order.

6.4.1 Processes and Channels

As presented in Section 2.2, we model a distributed system as a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ which are connected through pairwise bidirectional, reliable communication channels in a fully connected topology. In the following, we will also use p, q, r, etc. to denote processes. We denote the channel from p to q by c_{pq}.

6.4.2 Time Model

We assume that a local real-time clock is available to each process in the system, but clocks are not necessarily synchronized within the network. The system is assumed to be partially synchronous meaning that eventually, unknown bounds on all important network parameters (processing speed differences, message delivery delay) hold.

We also allow a minority of processes to be asynchronous. However, for our convenience, we model that asynchrony as timing failures (see next section).

6.4.3 Process Failures

We assume the general omission failure model. Recall that in the general omission model processes can fail by crashing or experience either send omissions or receive omissions. In our system we allow the possibility of *transient* omissions, i.e., a process may temporarily drop messages and later on reliably deliver messages again. Of course, *permanent* omissions are possible too. We also allow some processes to behave asynchronously.

Formally, processes can experience different kinds of failures: crash failures, omission failures and timing failures.

A *crash failure set* $F_c \subset \Pi$ is a subset of processes. Informally, F_c contains all processes that will eventually crash.

A *send-omission failure set* $F_{so} \subset \Pi \times \Pi$ is a relation over Π. Informally, $(p, q) \in F_{so}$ means that process p experiences at least one send omission towards q. If $(p, q) \notin F_{so}$ then p never experiences a send omission towards q.

Similarly, a *receive-omission failure set* $F_{ro} \subset \Pi \times \Pi$ is a relation over Π. Informally, $(p, q) \in F_{ro}$ means that process q experiences at least one receive omission from p. So if $(p, q) \notin F_{ro}$ then q never experiences a receive omission from p.

Some processes may experience timing failures. Timing failures refer to process asynchrony. We define an *asynchronous process failure set* $F_{ap} \subset \Pi$ as a subset of processes. Intuitively, F_{ap} contains all processes which are asynchronous in the system. Processes which are not in F_{ap} are eventually synchronous meaning that their processing speed is eventually bounded. Recall that a process p is *synchronous* if there exists a bound Φ such that the time between the execution of any two steps of p is bounded by Φ. A process p is *eventually synchronous* if there exists a time after which p is synchronous. Note that this implies that the relative process speeds between any pair of eventually synchronous processes is bounded. In our system model, both Φ and the time after which Φ holds are unknown.

A *process failure set* $F = (F_c, F_{so}, F_{ro}, F_{ap})$ is a tuple consisting of a crash failure set, a send-omission failure set, a receive-omission failure set, and an asynchronous process failure set.

6.4.4 Correct Processes

We define the set of *correct processes* to be the set of all processes that neither crash nor experience any omission nor are asynchronous. We denote this set with C. Formally:

$$C = \{p \in \Pi : p \notin F_c \wedge p \notin F_{ap} \wedge (\forall q \in \Pi : (p, q) \notin F_{so} \wedge (q, p) \notin F_{ro})\}$$

We assume that a majority of processes are correct, i.e., $|C| > \lfloor n/2 \rfloor$.

6.4.5 Send and Receive

Processes can send a message using the *Send* primitive. A $Send(p, m, q, t)$ event means that at time t process p sends m to q. In effect, m is inserted into the channel c_{pq} unless p experiences a send omission of m towards q. For any message m inserted into c_{pq}, the channel guarantees that eventually an appropriate event is added to the local event queue of process q. However, due to asynchrony or omissions of processes, there is not a time bound for the event in q to be added. In case q experiences a receive omission, m is removed from the channel without adding an appropriate event to the event queue of q. When this event is processed, we say that the message is received at time t, formalized as the occurrence of the event $Receive(p, m, q, t)$. We allow processes to selectively wait for messages.

Communication through a channel is reliable between correct processes, i.e., every message sent is eventually received and no message is received without having been

sent. Duplicated messages are discarded. Formally:

$$\forall p, q \in C : \forall m : \forall t_s \in T : Send(p, m, q, t_s) \\ \Rightarrow \exists t_r \in T : t_r > t_s : Receive(p, m, q, t_r)$$

and

$$\forall p, q \in \Pi : \forall m : \forall t_r \in T : Receive(p, m, q, t_r) \\ \Rightarrow \exists t_s \in T : t_r > t_s : Send(p, m, q, t_s)$$

No particular order relations are defined in the reception of messages. We assume that every message m from p to q is tagged with a unique *sequence number* assigned by p.

6.4.6 Channel Failures

Although we first assumed reliable channels (see Section 6.4.1), now we will model some of the previously presented process failures as channel failures.

We say that a message m from p to q is *received timely* if the receive event of m happens at most Ψ clock ticks after the send event of m, being Ψ a time bound. We say that the channel c_{pq} is *eventually timely* if there exists a point in time t such that all messages from p to q sent and received after t are received timely. Formally:

$$\exists t \in T : \exists\ bound\ \Psi : \forall m : Receive(p, m, q, t_r) \wedge \\ Send(p, m, q, t_s) \wedge t < t_s < t_r \Rightarrow (t_r - t_s \leq \Psi)$$

Again, in our partially synchronous system model both Ψ and the time after which Ψ holds are unknown.

We define an *asynchronous channel failure set* $F_{ac} \subset \Pi \times \Pi$ as a relation over Π such that $c_{pq} \in F_{ac}$ iff the channel c_{pq} is asynchronous. Note that if $c_{pq} \notin F_{ac}$ then c_{pq} is eventually timely. Note also that $c_{pq} \notin F_{ac}$ does not necessarily imply that $c_{qp} \notin F_{ac}$. However, we assume that every channel between correct processes is eventually timely, i.e., $\forall p, q \in C : c_{pq} \notin F_{ac}$.

Considering both process and channel failures, we define now the relation $\rightarrow$ to denote an *eventually timely and failure-free direct communication* from p to q:

$$p \rightarrow q \Leftrightarrow \begin{cases} p \notin F_c \wedge p \notin F_{ap} \wedge \\ q \notin F_c \wedge q \notin F_{ap} \wedge \\ (p, q) \notin F_{so} \wedge (p, q) \notin F_{ro} \wedge \\ c_{pq} \notin F_{ac} \end{cases}$$

Note that not necessarily $p, q \in C$. Clearly, the relation $\rightarrow$ is satisfied by every pair of correct processes, i.e., $\forall p, q \in C : p \rightarrow q$. The relation $\rightarrow$ is reflexive, not symmetric and not transitive.

6.4.7 Indirect Communication

Two processes p and q may communicate directly through the channel c_{pq}, or indirectly using message relaying through some path of any length. For example, consider the following relaying mechanism that enables indirect communication:

- To send a message m_{pq} at time t_s from p to q, p executes:

$$\forall r \in \Pi - \{p\} : \mathit{Send}(p, m_{pq}, r, t_s)$$

- The first time p executes the event $\mathit{Receive}(-, m_{rs}, p, t_r)$, if $p \neq s$ then p executes (δ denotes the local processing time for relaying):

$$\forall u \in \Pi - \{p, r\} : \mathit{Send}(p, m_{rs}, u, t_r + \delta)$$

We will address later how to achieve indirect communication without an explicit relaying mechanism.

6.4.8 Timely Communication

When considering indirect communication, a message from p could be received timely by q despite that the channel c_{pq} is not eventually timely. We first define the relation $\overset{*}{\rightarrow}$ to denote an eventually timely and failure-free (direct or indirect) communication from p to q. Informally, $p \overset{*}{\rightarrow} q$ means that there is a path from p to q such that for every adjacent processes r and s along the path, $r \rightarrow s$ is satisfied. Formally:

$$p \overset{*}{\rightarrow} q \Leftrightarrow (p \rightarrow q) \vee (\exists r : (p \rightarrow r \wedge r \overset{*}{\rightarrow} q))$$

The relation $\overset{*}{\rightarrow}$ is reflexive, not symmetric and transitive.

Observe that the relation $\overset{*}{\rightarrow}$ implies a *stable* eventually timely path from p to q. Observe also that the relation $\overset{*}{\rightarrow}$ does not impose any condition on the failure of the channel c_{pq}.

Nevertheless, a stable path is not actually required in order for q to receive messages timely from p. In fact, messages from p can be timely received by q as long as *at any time* (i.e., for each individual message) there exists a timely path from p to q, as we now explain.

We define now the weaker relation $\rightsquigarrow$, new as far as we know, to denote a failure-free (direct or indirect) communication from p to q such that eventually all messages from p to q are received timely by q. We will base all our definitions of timeliness on this notion. Formally:

$$\begin{aligned} p \rightsquigarrow q \Leftrightarrow\ & p \notin F_c \wedge q \notin F_c \wedge p \notin F_{ap} \wedge q \notin F_{ap} \\ & \wedge ((\forall m_{pq} : \forall t_s \in T : \mathit{Send}(p, m_{pq}, q, t_s)) \\ & \Rightarrow (((\exists t_r \in T : t_r > t_s : \exists\ \mathit{bound}\ \Psi_{m_{pq}} : \mathit{Receive}(-, m_{pq}, q, t_r)) \\ & \wedge \exists t \in T : t_s > t : t_r - t_s < \Psi_{m_{pq}})) \end{aligned}$$

It is important to note that, contrary to the relation $p \xrightarrow{*} q$, in $p \rightsquigarrow q$ the communication from p to q is not eventually timely in terms of a *global* bound that holds for every message. Instead, the bound $\Psi_{m_{pq}}$ only holds on a *per-message* basis, i.e., if for a given message m^i_{pq} a time bound $\Psi_{m^i_{pq}}$ holds, the next message m^{i+1}_{pq} will have a new, possibly higher bound $\Psi_{m^{i+1}_{pq}}$. This means that message delays from p to q could increase forever, which reflects the fact that not timely channels in a path may behave timely for some messages and not timely for others. This may seem to be a contradiction to the fact that failure detection is not implementable in totally asynchronous systems. However, observe that although delays may increase infinitely often, messages can be delivered without timing out on them. In a sense, this form of synchrony is more similar to the *progress assumptions* of the timed-asynchronous system model by *[Cristian and Fetzer, 1999]*.

The following relations hold:

$$p \xrightarrow{*} q \Rightarrow p \rightsquigarrow q$$

$$p \xrightarrow{*} q \wedge q \rightsquigarrow r \Rightarrow p \rightsquigarrow r$$

$$p \rightsquigarrow q \wedge q \xrightarrow{*} r \Rightarrow p \rightsquigarrow r$$

The relation $\rightsquigarrow$ is reflexive, not symmetric and transitive. Again, the relation $p \rightsquigarrow q$ does not impose any condition on the failure of the channel c_{pq}. We will use this relation to define the notion of connectedness on which the properties of the failure detector are based.

6.4.9 Connectedness

We define now the set of *in-connected processes* recursively as follows:

(1) Every correct process is in-connected, and

(2) a process p is in-connected if there exists an in-connected process q for which $q \rightsquigarrow p$.

Formally:

$$\textit{in-connected} = \{p \in \Pi : p \in C \vee \exists q \in C : q \rightsquigarrow p\}$$

Similarly, the set of *out-connected processes* is recursively defined as follows:

(1) Every correct process is out-connected, and

(2) a process p is out-connected if there exits an out-connected process q for which $p \rightsquigarrow q$.

Formally:

$$\textit{out-connected} = \{p \in \Pi : p \in C \vee \exists q \in C : p \rightsquigarrow q\}$$

6.4.10 Failure Detectors

The *range* of the failure detector is $\mathcal{R} = 2^{\Pi} \times \{\text{TRUE}, \text{FALSE}\}$. Informally it consists of a set of out-connected processes as well as a boolean flag indicating whether the process considers itself as in-connected.

A failure detector history H is a function $H : \Pi \times T \mapsto \mathcal{R}$. Intuitively, $H(p,t) = (\{p_1, p_2\}, \text{FALSE})$ for example means that at time t process p considers processes p_1 and p_2 as out-connected and does not consider itself as in-connected. We denote by H_1 and H_2 the projection of a tuple to its first or second value, i.e., $H_1(x,y) = x$ and $H_2(x,y) = y$.

A failure detector $\mathcal{D}$ is a function that maps a failure set F to a set of failure detector histories. Intuitively, these are all possible outputs which the failure detector can return.

Failure Detector Properties

A property of a failure detector is described by a set of failure detector histories. In this chapter we consider the following three properties:

- Strong Completeness. Every process that is not out-connected will not be permanently considered as out-connected by any in-connected process. Formally:

$$\forall F : \forall p \in \textit{in-connected} : \forall q \notin \textit{out-connected} : \\ \forall t_1 \in T : \exists t_2 \in T : t_2 > t_1 : q \notin H_1(p, t_2)$$

- Eventual Strong Accuracy. Eventually every out-connected process will be permanently considered as out-connected by every in-connected process. Formally:

$$\forall F : \forall p \in \textit{in-connected} : \forall q \in \textit{out-connected} : \\ \exists t_1 \in T : \forall t_2 \in T : t_2 > t_1 : q \in H_1(p, t_2)$$

- In-connectedness. Eventually every process will permanently consider itself as in-connected iff it is in-connected. Formally:

$$\forall F : \forall p \in \Pi : p \in \textit{in-connected} \Leftrightarrow \\ \exists t_1 \in T : \forall t_2 \in T : t_2 > t_1 : H_2(p, t_2) = \text{TRUE}$$

In our system model, we define the class of *eventually perfect* failure detectors, denoted $\Diamond\mathcal{P}$, as the set of all failure detectors satisfying strong completeness, eventual strong accuracy, and in-connectedness.

6.5 Failure Detector Algorithm

The failure detection algorithm presented in this section not only satisfies the properties defined in Section 6.4.10, but also builds a basis for the consensus algorithm of Section 6.6. Specifically, our failure detector determines the connectivity relations defined in Section 6.4.

The algorithm is based on heartbeat messages that every process sends periodically to the rest of processes. This schema provides a kind of delayed message forwarding, that enables indirect communication of information attached to heartbeat messages. At the same time, it provides the consensus layer with the support for piggybacking consensus level messages, as we will see in Section 6.7.

Algorithms 6.1, 6.2 and 6.3 present an algorithm implementing $\diamond\mathcal{P}$ according to the properties defined in Sect. 6.4.10. The algorithm provides to every process p a set of out-connected processes, $OutConnected_p$ and a boolean variable, $I\_am\_InConnected_p$. The set $OutConnected_p$ will eventually and permanently contain all the out-connected processes in case p is in-connected. With regard to the $I\_am\_InConnected_p$ variable, it will be TRUE if p is in-connected.

The algorithm is based on the periodical communication of heartbeat messages from every process to the rest of processes. Roughly speaking, heartbeat messages carry information about connectivity among processes. When a process p receives a heartbeat message, it uses the connectivity information carried by the message in order to update its perception of the connectivity of the rest of processes. This information, together with p's own connectivity, gives p a view of the current system connectivity, which will be propagated to the rest of processes attached to p's subsequent heartbeat messages. Next we explain in detail how our algorithm implements this approach.

Every message has associated a sequence number used to detect message omissions, and received messages are buffered to be delivered in FIFO order. Heartbeat messages carry a matrix M_p of $n \times n$ elements with connectivity information. The matrix M_p of a process p is updated according to the state of its input channels and with the matrix M_q received in the heartbeat message from a process q. This matrix will represent direct connectivity, i.e., $\rightarrow$ relations, between every pair of processes. Initially, all processes are supposed to be correct, so every element in the matrix has a value of 1. If all messages sent from a process q to a process p are received timely by p, the value of $M_p[p][q]$ will be maintained to 1. Otherwise, process p will set $M_p[p][q]$ to 0. Observe that in the algorithm a process does not monitor itself and, as a consequence, the elements of the main diagonal of the matrix are always set to 1.

Actually, M_p represents the transposed adjacency matrix, a (0,1)-matrix of a directed graph, where the value of the element $M_p[p][q]$ shows if there is an arc from q to p. This matrix will have the information needed to calculate the set of out-connected processes and the value of $I\_am\_InConnected_p$. From the definition of out-connectedness, a process q is out-connected if it communicates properly (i.e., timely and without omissions) with one correct process. From our failure assumptions a correct process communicates properly with at least $\lfloor n/2 \rfloor$ correct processes. It follows that an out-connected process

Algorithm 6.1: $\diamond\mathcal{P}$ in the omission model: main algorithm

```
1  Procedure main()
2      OutConnected_p ← Π
3      I_am_InConnected_p ← TRUE
4      foreach q ∈ Π − {p} do
5          Δ_p(q) ← default time-out interval          {Δ_p(q) → duration of p's time-out interval for q}
6          next_send_p[q] ← 1                          {sequence number of the next message sent to q}
7          next_receive_p[q] ← 1                       {sequence number of the next message expected from q}
8          Buffer_p[q] ← ∅
9      foreach q ∈ Π do
10         foreach u ∈ Π do M_p[q][u] ← 1               {M_p[q][u] = 0 means that q has not received at least one message from u}
11         Version_p[q] ← 0                            {Version_p contains the version number for every row of M_p}
12     UpdateVersion_p ← FALSE
13 || Task 1: repeat periodically                      {Sending heartbeats}
14     if UpdateVersion_p then                         {p's row has changed}
15         Version_p[p] ← Version_p[p] + 1
16         UpdateVersion_p ← FALSE
17     foreach q ∈ Π − {p} do
18         send (ALIVE, p, next_send_p[q], M_p, Version_p) to q      {sends a heartbeat}
19         next_send_p[q] ← next_send_p[q] + 1          {p updates its sequence number for q}
20 || Task 2: repeat periodically                      {Checking time-outs}
21     if ( p did not receive (ALIVE, q, next_receive_p[q], M_q, Version_q)
          from q ≠ p during the last Δ_p(q) ticks of p's clock ) then
                                                       {the next msg. in the sequence has not been received timely}
22         if M_p[p][q] = 1 then
23             Δ_p(q) ← Δ_p(q) + 1
24             M_p[p][q] ← 0                           {the potential omission is reflected in M_p}
25             UpdateVersion_p ← TRUE
26             call update_Connectivity()
27 || Task 3: when receive (ALIVE, q, c, M_q, Version_q) for some q      {Processing msgs. in order}
28     insert (ALIVE, q, c, M_q, Version_q) into Buffer_p[q]
29     while (ALIVE, q, next_receive_p[q], M_q, Version_q) ∈ Buffer_p[q] do      {next expected msg. from q}
30         call deliver_next_message(q, M_q, Version_q)      {msg. is delivered}
31         remove (ALIVE, q, M_q, next_receive_p[q], Version_q) from Buffer_p[q]
32         next_receive_p[q] ← next_receive_p[q] + 1
33     if Buffer_p[q] = ∅ then
34         M_p[p][q] ← 1                               {so far, p has received all msg. from q}
35         UpdateVersion_p ← TRUE
36     if M_p has changed then call update_Connectivity()
```

communicates properly with at least $\lfloor n/2 \rfloor$ processes. This communication is not necessarily direct communication ($\rightarrow$ relations), and therefore, the algorithm calculates powers of the adjacency matrix to find timely paths of any length between processes, which correspond to transitive relations $q \rightsquigarrow p$. The i-th power of an adjacency matrix gives us the number of paths of length i between processes. But remembering that in the matrix M_p the main diagonal is set to 1 instead of 0, in its n-th power, $A_p = (M_p)^n$, $A_p[p][q] \neq 0$ if there is a path of length equal or less to n from q to p. In fact, it is enough if we obtain the $\lceil (n+1)/2 \rceil$-th power of the adjacency matrix because in our

Algorithm 6.2: $\diamond\mathcal{P}$ in the omission model: procedure update_Connectivity()

Result: $OutConnected_p$ set and $I\_am\_InConnected_p$ boolean

```
37 Procedure update_Connectivity()
38   A_p ← (M_p)^n                                   {A_p is the n-th power of the M_p matrix}
39   foreach u, v ∈ Π do
40     if A_p[u][v] > 0 then A_p[u][v] ← 1
41   Out ← ∅
42   foreach q ∈ Π do
43     if (Σ_{i=0}^{n-1} A_p[i][q] ≥ ⌈(n+1)/2⌉) then Out ← Out ∪ {q}
44   OutConnected_p ← Out
45   I_am_InConnected_p = (Σ_{i=0}^{n-1} A_p[p][i] ≥ ⌈(n+1)/2⌉)
```

Algorithm 6.3: $\diamond\mathcal{P}$ in the omission model: procedure deliver_next_message()

Input: q: process from which the msg. has been received; M_q: q's knowledge about the system; $Version_q$: version number of each row of M_q

Result: update of M_p matrix and $Version_p$ vector

```
46 Procedure deliver_next_message()
47   foreach v ∈ Π do M_p[q][v] ← M_q[q][v] ;        {q's row of M_q is copied into M_p}
48   Version_p[q] ← Version_q[q]
49   foreach u ∈ Π − {p, q} do
50     if Version_q[u] > Version_p[u] then           {q's information about u is more recent than p's}
51       foreach v ∈ Π do M_p[u][v] ← M_q[u][v]
52       Version_p[u] ← Version_q[u]
```

system model the distance among all correct processes is 1, and therefore, the maximum graph's diameter is $\lceil (n+1)/2 \rceil$.

The set of out-connected processes, along with the $I\_am\_InConnected_p$ variable, is computed in the *update_Connectivity*() procedure (see Algorithm 6.2), which is called every time a value of the matrix M_p is changed. Observe that it is important for a process p to check its own in-connectivity to verify the validity of the information contained in M_p. A process p is in-connected if more than $\lfloor n/2 \rfloor$ processes communicate properly with it. This condition is checked in the *update_Connectivity*() procedure too and its value is output to the $I\_am\_InConnected_p$ variable.

In the algorithm, every process p executes three tasks:

- In Task 1 (line 13), p periodically sends a heartbeat message to the rest of processes. Remember that every message is tagged with a unique sequence number. The matrix M_p is sent within the heartbeat messages.
- In Task 2 (line 20), if p does not receive the expected message from a process q (according to the $next\_receive_p[q]$ sequence number) in the expected time, the value of $M_p[p][q]$ is set to 0.
- In Task 3 (line 27), received messages are processed. The messages p receives from another process q are inserted in a FIFO buffer $Buffer_p[q]$ (line 28), and delivered following the sequence number $next\_receive_p[q]$. Once delivered the next expected

message from a process q, the condition of empty buffer means that there is no message left from q, so $M_p[p][q]$ is set to 1.

The procedure *deliver_next_message*() (Algorithm 6.3) is used to update the adjacency matrix M_p using the information carried by the message. In the procedure, process p copies into M_p the row q of the matrix M_q received from q. This way, p learns about q's input connectivity. With respect to every other process u, a mechanism based on version numbers is used to avoid copying old information about u's input connectivity. Process p will only copy into M_p the row u of M_q if its associated version number is higher.

6.5.1 Correctness Proof

We now show that the algorithm of Algorithm 6.1, 6.2 and 6.3 implements $\diamond\mathcal{P}$ in the omission model.

Observation 6.1: Observation 1*: At every process p, the matrix M_p is updated with its own connectivity information and with the matrices M_q received in the heartbeat messages. The updated M_p and its version number $Version_p$ are sent with p's next heartbeat message. The local delay in process p to send M_p and $Version_p$ is bounded in the algorithm by the period of Task 1 of p, which is finite if p is eventually synchronous and has not crashed. This way, a schema equivalent to the relaying mechanism described in Section III is obtained.*

Lemma 6.1: *$\forall p, q \in \Pi$, iff $q \rightsquigarrow p$, then $(M_p)^n[p][q] \geq 1$ eventually and permanently.*

If $q \rightsquigarrow p$, then, by definition of $\rightsquigarrow$, eventually and permanently there is a timely path with no omission of any length from q to p. By Task 1 q sends periodically a heartbeat message including its updated M_q and $Version_q[q]$ to the rest of processes. Every process r receiving by Task 3 a new $Version_q[q]$ will update from M_q, in the procedure *deliver_next_message*(), the row q of M_r as well as $Version_r[q]$. Process r will update matrix M_r too with its own connectivity information: by Task 3 of r, $M_r[r][q]$ is set to 1 every time $Buffer_r[q]$ becomes empty; by Task 2 $M_r[r][q]$ is set to 0 when the next expected message from q is not received timely by r. By Observation 6.1, $M_r[r][q]$ and $Version_r[r]$ are propagated to every process s if s is eventually synchronous and has not crashed. (Note that, as a particular case, r or s may be p.) When some message is delivered by Task 3 of p, by the procedure *deliver_next_message*(), p will update M_p and calculate $(M_p)^n$ if some value in M_p has changed. By the definition of the relation $\rightsquigarrow$, If $q \rightsquigarrow p$ then $(M_p)^n[p][q]$ will be evaluated eventually and permanently to a positive value, otherwise, if not $q \rightsquigarrow p$, $(M_p)^n[p][q]$ will not be set to 1 permanently because, by definition, in every possible path from q to p there will be two processes, r and s, such that not $r \rightarrow s$ (r and s could be q and p), and therefore, $(M_p)^n[p][q] = 0$.

Lemma 6.2: *$\forall p \in$ in-connected, $\forall q \in$ out-connected, eventually and permanently $q \in OutConnected_p$.*

Proof. By definition of out-connected process, there is some correct process r such that $q \rightsquigarrow r$. By definition of in-connected process, there is some correct process s

such that $s \rightsquigarrow p$. By transitivity, for every correct process u, $q \rightsquigarrow u$, and in particular $q \rightsquigarrow s$. By Lemma 6.1, $(M_s)^n[s][q] \geq 1$. By the procedure *deliver_next_message*(), s will update M_s copying all the rows from at least the rest of correct processes. As a consequence of that, since $|C| > \lfloor n/2 \rfloor$, column q of $(M_s)^n$ will include eventually and permanently more than $\lfloor n/2 \rfloor$ positive values. Again by Lemma 6.1 and the procedure *deliver_next_message*(), column q of $(M_p)^n$ will have eventually and permanently a positive value for more than $\lfloor n/2 \rfloor$ processes. As a consequence, according to the procedure *update_Connectivity*(), q will be permanently included in the set $OutConnected_p$. □

Lemma 6.3: *$\forall p \in$ in-connected, $\forall q \notin$ out-connected, q is not permanently in $OutConnected_p$.*

Proof. Since q is not out-connected, it does not exist a correct process r such that $q \rightsquigarrow r$. By Lemma 6.1, $(M_p)^n[r][q] \geq 1$ only when q behaves timely with respect to r, however, since q is not out-connected, this will not occur permanently. As a consequence, since $|C| > \lfloor n/2 \rfloor$, the number of processes s such that $(M_p)^n[s][q] \geq 1$ will not be permanently greater than $\lfloor n/2 \rfloor$, and by the procedure *update_Connectivity*(), q will not be permanently included in the set $OutConnected_p$. □

Lemma 6.4: *$\forall p \in \Pi$, iff $p \in$ in-connected, then eventually and permanently $I\_am\_InConnected_p =$* TRUE.

Proof. If $p \in$ *in-connected*, $\forall r \in C$, $r \rightsquigarrow p$. By Lemma 6.1, and following a similar reasoning to the proof of Lemma 6.2, now applied to row p of $(M_p)^n$, iff $p \in$ *in-connected* the procedure *update_Connectivity*() will eventually and permanently set $I\_am\_InConnected_p$ to TRUE (line 45). □

Theorem 6.1: *The algorithm of Algorithm 6.1, 6.2 and 6.3 implements $\diamond\mathcal{P}$ in the omission model.*

Proof. The strong completeness, eventual strong accuracy, and in-connec-ti-vity properties of $\diamond\mathcal{P}$ are satisfied by Lemmas 6.3, 6.2, and 6.4 respectively. □

6.6 Consensus

We now focus our attention on the consensus layer of our architecture (see Figure 6.3), in which we implement consensus by using the failure detector class $\diamond\mathcal{P}$ defined in Section 6.4.10. The consensus algorithm itself is asynchronous, meaning that it tolerates arbitrary phases of asynchrony and delegates all timing aspects to the underlying failure detector. We first define the consensus problem, then give a consensus algorithm and prove its correctness. Finally, we give an example of an execution of the algorithm.

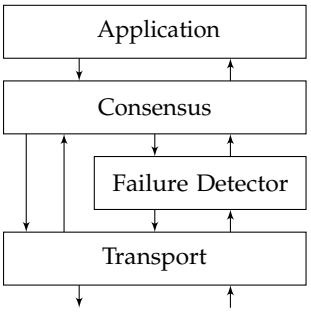


Figure 6.3: Architecture of the system

6.6.1 Reformulation of Properties

Recall from Chapter 3 that in the consensus problem, every process proposes a value, and correct processes must eventually decide on some common value that has been proposed. In the crash model, every correct process is required to eventually decide some value. This is called the termination property of consensus. In order to adapt consensus to the omission model, we argue that only the termination property has to be redefined. This property involves now every in-connected process, since, despite they can experience some omissions, in-connected processes are those that will be able to receive the decision.

The properties of consensus in the omission model are thus the following:

- C-Termination. Every in-connected process eventually decides some value.
- C-Integrity. Every process decides at most once.
- C-Uniform agreement. No two processes decide differently.
- C-Validity. If a process decides v, then v was proposed by some process.

6.6.2 Consensus Algorithm

Algorithms 6.4 and 6.5 present an algorithm solving consensus in the omission model using $\diamond\mathcal{P}$. It is an adaptation of the algorithm proposed by *[Chandra and Toueg, 1996]*, included in Appendix A.

The use of $\diamond\mathcal{P}$ by every process p is modeled by means of the following two variables: a boolean variable $I\_am\_InConnected_p$ which provides the in-connectedness property, and a set $OutConnected_p$ which provides the strong completeness and eventual strong accuracy properties.

For brevity, instead of explaining the algorithm from scratch, we just comment on the modifications required to adapt the original algorithm:

Algorithm 6.4: Solving consensus in the omission model using $\diamond\mathcal{P}$: main algorithm

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

  {Rotate through coordinators until decision is reached}
  while $state_p = undecided$ **do**
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ {c_p is the current coordinator}
    Phase 1: {All processes p send $estimate_p$ to the current coordinator}
      send $(p, r_p, estimate_p, ts_p)$ to c_p
    Phase 2: { The current coordinator tries to gather $\lceil \frac{(n+1)}{2} \rceil$ estimates. If it succeeds, it proposes a new estimate. Otherwise, it sends a NEXT message to all }
    if $p = c_p$ **then**
      wait until (**not** ($I\_am\_InConnected_p$) **or** (for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ from q))
      if *for* $\lceil \frac{(n+1)}{2} \rceil$ *processes q: received* $(q, r_p, estimate_q, ts_q)$ *from q* **then**
        $success_p \leftarrow$ TRUE
        $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
        $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
        $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
        send $(p, r_p, estimate_p)$ to all
      else
        $success_p \leftarrow$ FALSE
        send $(p, r_p,$ NEXT$)$ to all
    Phase 3: {All processes wait for the new estimate proposed by the coordinator}
    wait until (**not** ($I\_am\_InConnected_p$) **or** ($c_p \in \Pi - OutConnected_p$) **or** received [$(c_p, r_p, estimate_{c_p})$ **or** $(c_p, r_p,$ NEXT$)$] from c_p)
    if *received* $(c_p, r_p, estimate_{c_p})$ *from* c_p **then**
      $estimate_p \leftarrow estimate_{c_p}$
      $ts_p \leftarrow r_p$
      send (p, r_p, ack) to c_p
    else
      send $(p, r_p, nack)$ to c_p
    Phase 4: { If the current coordinator sent a valid estimate in Phase 2, it waits for replies of out-connected processes while it considers itself as in-connected. If $\lceil \frac{(n+1)}{2} \rceil$ processes replied with *ack*, the coordinator R-broadcasts a decide message }
    if $(p = c_p)$ **and** $(success_p =$ TRUE$)$ **then**
      wait until [**not** ($I\_am\_InConnected_p$) **or** for every process q: (*received* (q, r_p, ack) **or** *received* $(q, r_p, nack)$ **or** $q \in \Pi - OutConnected_p$)]
      if *for* $\lceil \frac{(n+1)}{2} \rceil$ *processes q:received* (q, r_p, ack) **then** *R-broadcast*$(p, r_p, estimate_p,$ DECIDE$)$

Algorithm 6.5: Solving consensus in the omission model using $\diamond\mathcal{P}$: deciding

```
{If p R-delivers a decide message, p decides accordingly}
35 when R-deliver(q, r_q, estimate_q, DECIDE) do
36   if state_p = undecided then
37     decide(estimate_q)
38     state_p ← decided
```

- In Phase 2, the current coordinator waits for a majority of estimates while it considers itself as in-connected in order not to block. Only in case it receives a majority of estimates a valid estimate is sent to all. If it is not the case, the coordinator sends a NEXT message indicating that the current round cannot be successful.
- In Phase 3, every process p waits for the new estimate proposed by the current coordinator while p considers itself as in-connected and the coordinator as out-connected in order not to block. Also, p can receive a NEXT message indicating that the current round cannot be successful. In case p receives a valid estimate, p adopts it and replies with an *ack* message. Otherwise, p sends a *nack* message to the current coordinator.
- In Phase 4, if the current coordinator sent a valid estimate in Phase 2, it waits for replies of out-connected processes while it considers itself as in-connected in order not to block. If a majority of processes replied with *ack*, the coordinator broadcasts a decide message using a Reliable Broadcast primitive *[Hadzilacos and Toueg, 1994]* (see Appendix B).

When a process p sends a message m to another process q, the following relaying approach is assumed: (1) p sends m to all processes, including q, except p itself, and (2) whenever p receives for the first time a message m whose actual destination is another process q, p forwards m to all processes (except the process from which p has received m and p itself). This approach can take advantage of the underlying all-to-all implementation of the $\diamond\mathcal{P}$ failure detector, as we will see in the next section.

6.6.3 Correctness Proof

Since the algorithm is very similar to the one proposed by *[Chandra and Toueg, 1996]*, we only sketch the correctness proof here. First of all, observe that uniform agreement is preserved, because we keep the original mechanism based on majorities to decide on a value. Also, it is easy to see that integrity and validity are satisfied. Finally, in order to show that termination is satisfied, we first show that the algorithm does not block in any of its *wait* instructions:

- In Phase 2, if the current coordinator p is not in-connected, it will eventually stop waiting because the failure detector will eventually set the variable $I\_am\_InConnected_p$ to FALSE. On the other hand, if p is in-connected, it will eventually receive a majority

of estimates since by definition there is a majority of correct processes in the system. Hence, no coordinator blocks forever in the *wait* instruction of Phase 2.

- In Phase 3, every process p waits for the new estimate proposed by the current coordinator or a NEXT message while p considers itself as in-connected and the coordinator as out-connected. Clearly, by the properties of $\diamond\mathcal{P}$ no process blocks forever in the *wait* instruction of Phase 3.
- In Phase 4, the current coordinator waits for replies of out-connected processes while it considers itself as in-connected. Again, by the properties of $\diamond\mathcal{P}$ no coordinator blocks forever in the *wait* instruction of Phase 4.

By the previous facts, eventually some correct process c will coordinate a round in which:

- In Phase 2, the coordinator c will receive a majority of estimates, because $I\_am\_InConnected_c$ will be permanently set to TRUE (by the properties of $\diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will send a valid estimate to all processes at the end of Phase 2.
- In Phase 3, every correct process p will receive c's valid estimate, because $I\_am\_InConnected_p$ will be permanently set to TRUE and c will be permanently in $OutConnected_p$ (by the properties of $\diamond\mathcal{P}$). Hence, p will send an *ack* message to c at the end of Phase 3.
- In Phase 4,the coordinator c will receive a majority of *ack* messages, because $I\_am\_InConnected_c$ will be permanently set to TRUE and all correct processes will be permanently in $OutConnected_c$ (by the properties of $\diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will R-broadcast the decision, and every in-connected process will eventually decide.

6.6.4 Example of Execution

Figure 6.4 shows an example of an execution of the consensus algorithm. Observe that the decision can be a value proposed by a non-correct process, e.g., an exclusively out-connected process like p_2 in the example. Observe also that exclusively in-connected processes like p_5 are able to deliver the decision (and hence to decide).

6.7 Integrating Failure Detection and Consensus

In the system there are two types of messages sent over the channel: consensus algorithm messages and failure detector messages. We call the former *protocol messages* and the latter *heartbeats*. Heartbeats are time critical, i.e., they should not be delayed by the transport layer, while protocol messages are asynchronous, i.e., eventual delivery is sufficient for them.

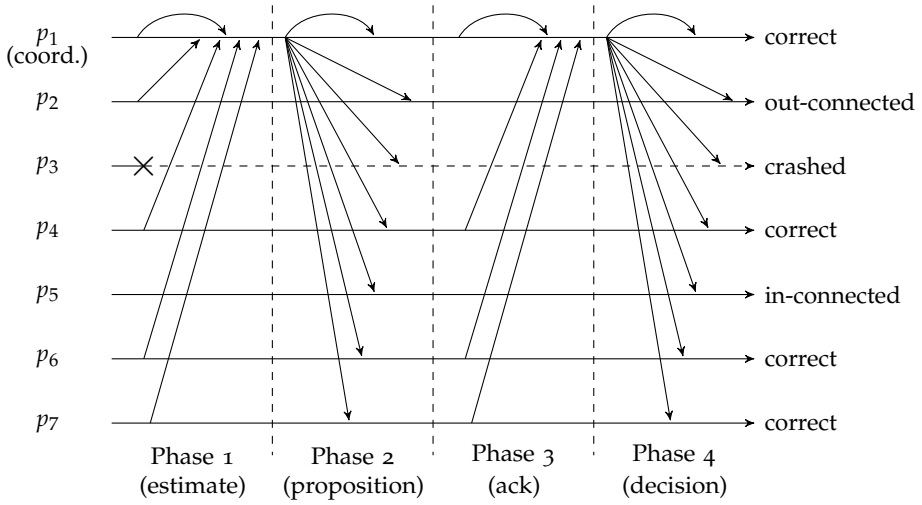


Figure 6.4: Example of an execution of the consensus algorithm

The system should be able to detect omissions of both types of messages. In the general omission model the implementation of the failure detector cannot be independent of the messages sent by the algorithm using it, as concluded by *[Doudou et al., 1999]*. Indeed, in this model incorrect processes could arbitrarily stop sending consensus messages while they continue sending failure detector messages. To cope with such a behaviour, we propose to insert consensus messages into failure detector messages, as it is done in *[Cortiñas et al., 2007]*. A different approach, followed in *[Delporte-Gallet, Fauconnier, Tielmann, Freiling, and Kilic, 2009]*, consists in abandoning the modular design provided by failure detectors, using timers directly in the consensus algorithm.

We propose to integrate both of them so that only one control point is needed. That is, we propose to insert consensus messages content into messages of the failure detector, in order to cope with omissions in the consensus algorithm. This way, heartbeats turn into the *transport mechanism* for protocol messages. Every heartbeat has a small fixed-size message field called the *payload*. Similar to network transport protocols (like IP), protocol messages are inserted into the payload of heartbeats when they are sent. If a protocol message is larger than the size of the payload it is fragmented into smaller parts which are sent one after the other. Similar to the fragmentation mechanism in IP, unique identifiers and sequence numbers allow to piece together the fragments at the destination in the correct order. If there is no protocol message to be sent, the payload of a heartbeat can remain *empty*.

We assume a *scrambler* which receives the protocol and failure detector messages and outputs equal looking messages of the same size in regular time intervals (see Figure 6.5). It proceeds as follows. Whenever a protocol message has to be sent, it will be piggybacked on the failure detector message. If there is no protocol message ready to be sent, the packet's payload will be filled with random bits. In order to be efficient, the predefined size of the messages sent will be kept as small as possible. If a

protocol message is too big, it will be divided, using a fragmentation mechanism, and piggybacked into multiple failure detector messages. Since the protocol is asynchronous, even long delays can be tolerated as long as the failure detector works correctly.

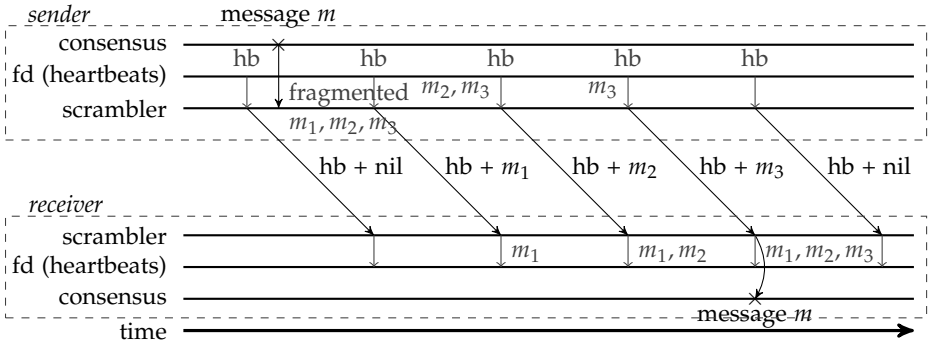


Figure 6.5: Example of scrambler's function

Observe that, following the proposed mechanism, the communication-pattern between every two processes is always the same, as long as there is no omissions or crashes, which is interesting for its application in the security area, as we will show in the next chapter.

6.8 Conclusions

In this chapter we have shown some issues related to failure detection in the general omission failure model. More precisely, we have presented an impossibility proof that shows that it is impossible to achieve the classical completeness and accuracy properties of $\diamond\mathcal{P}$ in the general omission failure model, so we propose a redefinition of the properties of failure detectors, based on the connectedness of the processes instead of their correctness.

We have modeled *connectedness* of processes according to in- and out-connected classes of processes and given a novel definition of the eventually perfect failure detector class for the general omission failure model. The new failure detector properties are based on process connectedness rather than on process correctness. It is remarkable that the failure detector can be used to solve consensus with very weak synchrony assumptions and a majority of good processes in the system. Interestingly, the consensus algorithm using the failure detector is a quite simple adaptation of the classical algorithm for the crash model of *[Chandra and Toueg, 1996]*.

Our algorithms can be improved with respect to communication efficiency. In particular, our implementation of $\diamond\mathcal{P}$ can be modified such that it results in a failure detector which could be similar to Ω by omitting the all-to-all message exchange pattern, saving a substantial amount of messages. However, the decision to choose $\diamond\mathcal{P}$ was deliberate since we were looking for an all-to-all communication pattern for security reasons, as

we will see in Chapter 7. Therefore, any such efficiency improvement would be futile in practical systems.

On the theoretical side it would be interesting to study the minimal storage and communication effort necessary to solve consensus in our model, since we use unbounded buffers in our implementation and the bit complexity of the messages we use is also rather high.

In the next chapter we will show how the Byzantine failure model can be reduced to the general omission failure model by using a secure platform named *TrustedPals*. Hence, the failure detector proposed in this chapter can be used as a building block to solve Secure Multiparty Computation, a problem that can be reduced to consensus.

Chapter 7

Solving Secure Multiparty Computation in the Byzantine Failure Model Using Failure Detectors in General Omission

In the previous chapter we presented a failure detector class for the general omission failure model. In this chapter we will show how that failure detector class, combined with a platform called *TrustedPals*, can be useful to solve *Secure Multiparty Computation*, SMC, a problem equivalent to consensus, in a Byzantine environment.

Informally, SMC allows a set of processes to compute a common function with certain security assumptions, even in a Byzantine environment. Although SMC has been proven to be solvable in synchronous systems, there is not a clear solution for asynchronous systems. We will show that SMC is solvable in asynchronous systems by using the failure detector class presented in the previous chapter.

As we will see, consensus and SMC are equivalent in our system. Additionally, we will study how TrustedPals allows a transformation of the system in the sense of *[Veríssimo, 2006]*; TrustedPals transforms the original Byzantine system, which we call *untrusted system*, into a *trusted system* with a stronger failure model, the general omission failure model, together with some asynchrony. We will show that the trusted system matches the system model we assumed in the previous chapter, and hence the failure detector presented in the previous chapter can be used to solve consensus and thus SMC.

Outline. This chapter is structured as follows. In Section 7.1 we introduce SMC and TrustedPals. Section 7.2 describes TrustedPals architecture, motivation and system model. More precisely, in Section 7.2.1 we fully formalize the system model of TrustedPals; in Section 7.2.3 we show the system model of the untrusted system and in Section 7.2.4 we present the system model of the trusted system. In Section 7.3 we describe how to integrate failure detection and consensus securely in the TrustedPals framework. Finally, we conclude in Section 7.4.

7.1 Introduction to Secure Multiparty Computation and TrustedPals

Consider a set of parties who wish to correctly compute some common function F of their local inputs, while keeping their local data as private as possible, but who do not trust each other, nor the channels by which they communicate. This is the problem of *Secure Multiparty Computation* (SMC) *[Yao, 1982]*. SMC is a very general security problem, i.e., it can be used to solve various real-life problems such as distributed voting, private bidding and online auctions, sharing of signature or decryption functions and so on. Unfortunately, solving SMC is — without extra assumptions — very expensive in terms of communication (number of messages), resilience (amount of redundancy) and time (number of synchronous rounds).

TrustedPals, proposed by *[Benenson, Fort, Freiling, Kesdogan, and Penso, 2006a]*, is a smartcard-based implementation of SMC which allows much more efficient solutions to the problem. Conceptually, TrustedPals considers a distributed system in which processes are locally equipped with tamper-proof security modules (see Figure 7.1). In practice, processes are implemented as a Java desktop application and security modules are realized using Java Card Technology enabled smartcards *[Chen, 2000]*, tamper-proof Subscriber Identity Modules (SIM) *[Leavitt, 2005]* like those used in mobile phones, or storage cards with builtin tamper-proof processing devices *[certgate GmbH, 2008]*. Roughly speaking, solving SMC among processes is achieved by having security modules that jointly simulate a *Trusted Third Party* (TTP), as we now explain.

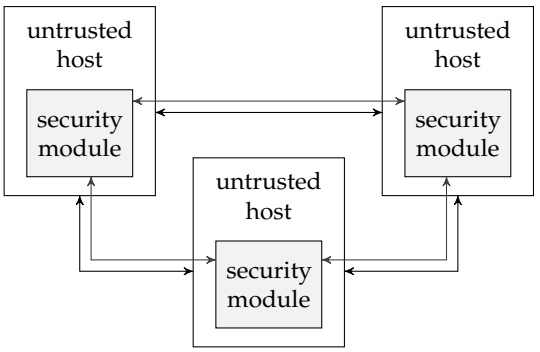


Figure 7.1: Processes with tamper-proof security modules

To solve SMC in the TrustedPals framework, the function F is coded as a Java function and is distributed within the network in an initial setup phase. Then processes hand their input value to their security module and the framework accomplishes the secure distribution of the input values. Finally, all security modules compute F and return the result to their process. The network of security modules sets up confidential and authenticated channels between each other and operates as a *secure overlay* within the distribution phase. Roughly speaking, within this secure overlay arbitrary and malicious behaviour of an attacker is reduced to rather benign faulty behaviour (process crashes and message omissions). TrustedPals therefore allows to reduce the security problem of SMC to a problem of fault-tolerant synchronization *[Benenson et al., 2006a]*,

an area which has a long research tradition in fault-tolerant distributed computing (see for example *[Lynch, 1996]*). However, solving the synchronization problem alone is not trivial, especially since we investigate it under message omission failures, a failure scenario which is rather unusual. Furthermore, the reduction from security to fault-tolerance creates a new set of validation obligations regarding the *integration* of a fault-tolerant algorithm into a secure system, which is also far from being trivial.

To date, TrustedPals and its implementation assumed a *synchronous* network setting, i.e., a setting in which all important timing parameters of the network are bounded and known. This makes TrustedPals sensitive to unforeseen variations in network delay and therefore not very suitable for deployment in networks like the Internet. In this chapter, we explore how to make TrustedPals applicable in environments with less synchrony. More precisely, we explore the possibilities to implement TrustedPals in a modular fashion inspired by results in fault-tolerant distributed computing: we use an *asynchronous* consensus algorithm and encapsulate (some very weak) timing assumptions, within a failure detector.

Related Work

Our work on TrustedPals can be regarded as building failure detectors for arbitrary (*Byzantine*) failures which has been investigated previously (see for example *[Kihlstrom, Moser, and Melliar-Smith, 2003; Doudou, Garbinato, and Guerraoui, 2002; Haeberlen, Kouznetsov, and Druschel, 2006]*). In contrast to previous works on Byzantine failure detectors, we use security modules to avoid the tar pits of this area. This contrasts TrustedPals to the large body of work that tackle Byzantine faults *directly*, "Practical Byzantine Fault-Tolerance" by *[Castro and Liskov, 2002]* or more recently the *Aardvark* system by *[Clement, Wong, Alvisi, Dahlin, and Marchetti, 2009]*. While being conceptually simpler, Byzantine-tolerant protocols necessarily have to assume a two-thirds majority of benign processes in non-synchronous settings *[Bracha and Toueg, 1985]* while TrustedPals needs only a simple majority. Next to an improved resilience, TrustedPals by design can provide *secrecy* of data against attackers, a notion that can only be achieved in Byzantine-tolerant algorithms by applying complex secret sharing mechanisms *[Herlihy and Tygar, 1987]*. All these advantages result from using security modules to constrain the attacker in such a way that Byzantine faults are reduced to general omission faults.

[Delporte-Gallet et al., 2005] were the first to investigate non-synchronous settings in the TrustedPals context, always with the (implicit) motivation to make TrustedPals more practical. Following the approach of *[Chandra and Toueg, 1996]* (and similar to this work) they separate the trusted system into an asynchronous consensus layer and a partially synchronous failure detection layer. They assume that transient omissions are masked by a piggybacking scheme. The main difference however is that they solve a *different version of consensus* than we do: roughly speaking, message omissions can cause processes to only be able to communicate *indirectly* and we admit processes to participate in consensus even if they cannot communicate directly. *[Delporte-Gallet et al.,*

2005] only guarantee that all processes that can communicate *directly* with each other solve consensus. In contrast, we allow also another set of processes to propose and decide: those which are able to send and receive messages even if it is indirectly. As a minor difference, we focus on the class $\diamond\mathcal{P}$ of eventually perfect failure detectors whereas they *[Delporte-Gallet et al., 2005]* implement the Ω failure detector. Furthermore, they *[Delporte-Gallet et al., 2005]* do not describe how to integrate failure detection and consensus within the TrustedPals framework: a realistic adversary who is able to selectively influence the communication messages of the algorithms for failure detection and consensus can cause their consensus algorithm to fail. This problem is partly addressed in a recent paper *[Delporte-Gallet et al., 2009]* where consensus and failure detection are integrated for efficiency purposes, not for security.

The original work on TrustedPals *[Benenson et al., 2006a]* reduced the problem of SMC to that of *Uniform Interactive Consistency* (UIC) in the trusted system. *[Raynal, 2005]* showed that UIC can be reduced to a perfect failure detector and so cannot be implemented in systems that only provide weaker timing assumptions like we do in this work. We reflect this fact by considering a variant of SMC that is reducible to uniform consensus (see Sect. 7.2).

Recently, solving SMC *without* security modules has received some attention focusing on two-party protocols *[MacKenzie, Oprea, and Reiter, 2003; Malkhi, Nisan, Pinkas, and Sella, 2004; Kolesnikov, 2005; Kruger, Jha, Goh, and Boneh, 2006; Lindell and Pinkas, 2007]*. In systems *with* security modules, *[Avoine and Vaudenay, 2003]* examined the approach of jointly simulating a TTP. This approach was later extended by *[Avoine, Gärtner, Guerraoui, and Vukolic, 2005]* who show that in a system with security modules fair exchange can be reduced to a special form of consensus. They derive a solution to fair exchange in a modular way so that the agreement abstraction can be implemented in diverse manners. *[Benenson et al., 2006a]* extended this idea to the general problem of SMC and showed that the use of security modules cannot improve the resilience of SMC but enables more efficient solutions for SMC problems. All these papers assume a *synchronous* network model.

[Correia, Veríssimo, and Neves, 2002] present a system which employs a real-time distributed security kernel to solve SMC. The architecture is very similar to that of TrustedPals as it also uses the notion of architectural hybridization *[Sousa, Neves, Veríssimo, and Sanders, 2006]*. However, the adversary model of *[Correia et al., 2002]* assumes that the attacker only has remote access to the system while TrustedPals allows the owner of a security module to be the attacker. Like other previous works *[Avoine et al., 2005; Avoine and Vaudenay, 2003; Benenson et al., 2006a]*, *[Correia et al., 2002]* also assume a synchronous network model at least in a part of the system.

Contributions

In this chapter we present a modular redesign of TrustedPals using consensus and failure detection as modules. More specifically, we make the following technical contributions:

- We show how to solve Secure Multiparty Computation by implementing TrustedPals in systems with some weak synchrony assumptions.
- We show how a Byzantine failure model can be transformed into a general omission one with some asynchrony level.
- We make use of the failure detector and consensus algorithms presented in Chapter 6 as a solution in this scenario.
- We integrate failure detection and consensus securely in TrustedPals by employing message padding and dummy traffic, tools known from the area of privacy enhancing techniques. In particular, all the communication happens at the failure detection level, where fixed size messages are sent periodically.

7.2 TrustedPals Overview: System Models

We now give an overview over the TrustedPals system architecture. This section is meant as an informal introduction giving a high level view of the definitions used in the remainder of the chapter.

7.2.1 Untrusted and Trusted System

We formalize the system assumptions within a hybrid model, i.e., the model is divided into two parts (see Figure 7.2). The lower part consists of n processes which represent the *untrusted hosts*. The upper part equally consists of n processes which represent the *security modules*. Because of the lack of mutual trust among untrusted hosts, we call the former part the *untrusted system*. Since the security modules trust each other we call the latter part the *trusted system*.

The processes in the untrusted system (i.e., the hosts) execute (possibly untrustworthy) user applications like e-banking or e-voting programs. Because of the untrustworthy nature of these processes, they use the trusted system as a subsystem to solve the involved security problems. The trusted system executes only programs which have been certified by some accepted authority. It is not possible for the user to install arbitrary software on the trusted system. The tamper-proof nature of the trusted processes allows to protect stored and transmitted information even from the untrusted processes on which they reside. The authority can be an independent service provider (like a network operator) and is only necessary within the *bootstrap* phase of the system, not during any *operational* phases (like running the SMC algorithms).

Formally, the connection between the untrusted and trusted system is achieved by associating each process in the untrusted system (i.e., each host) with exactly one process in the trusted system (i.e., a security module) and vice versa. This means that there exists a bijective mapping between processes in the untrusted system and processes in the trusted system. Since host and security module reside on the same physical machine, we assume that for each association there is a bidirectional synchronous communication

link, e.g., implemented by shared variables or message passing communication in the host operating system.

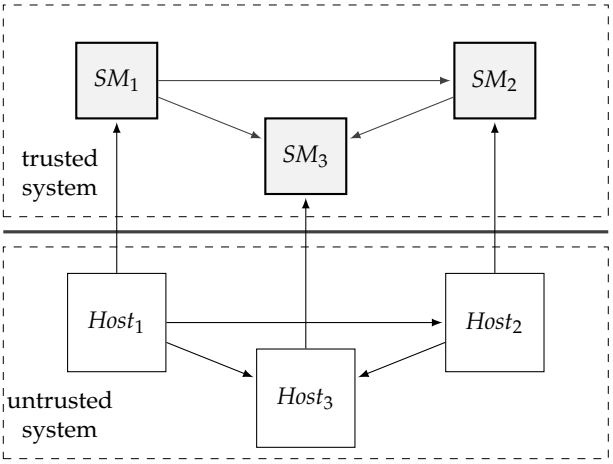


Figure 7.2: The untrusted and trusted system

Summarizing, there are two different types of processes: processes in the untrusted system and processes in the trusted system. For brevity, we will use the unqualified term *process* if the type of process is clear from the context.

7.2.2 Solving SMC with TrustedPals

Let $x_1, \ldots, x_n$ be the private inputs of each host. In SMC, the result $r = F(x_1, \ldots, x_n)$ of applying F to the private inputs should be computed reliably and securely, i.e., as if they were using a trusted third party (TTP). This means that the individual inputs remain secret to other processes (apart from what is given away by r) and that malicious processes can neither prevent the computation from taking place nor influence r in favorable ways.

In this chapter, we use the following definition of SMC: A protocol solves *secure multi-party computation (SMC)* with TrustedPals if it satisfies the following properties *[Goldreich, 2002; Benenson et al., 2006a]*:

- SMC-Validity. If a process receives a F-result, then F was computed on inputs of processes (and not on any other values).
- SMC-Agreement. No two values returned by TrustedPals differ.
- SMC-Termination. Every non-faulty process eventually receives a result by TrustedPals.
- SMC-Privacy. Faulty processes learn nothing about the input values of correct processes (apart from what is given away by the result r and the input values of all faulty processes).

Recall that these properties abstractly specify what happens when using a TTP *[Goldreich, 2002]*. There, the TTP waits for the inputs of all n processes and computes the value of F on all those inputs which it received. After computing F, the TTP sends the result back to all processes. SMC-Privacy is motivated by the fact that the TTP does all the processing and channels to the TTP are confidential: no information about other processes' input values leaks from this idealized entity, apart of what the result of F gives away when it is finally received by the processes. For example, if F computes the maximum of all input values, then the result r gives away that r was the input of *some* process. Note also that SMC-Privacy protects correct processes from cheating by faulty processes and therefore can also be understood as a fairness requirement (in the sense of fair exchange *[Pagnia, Vogt, and Gärtner, 2003]*).

To solve SMC in TrustedPals, untrusted processes delegate the confidential exchange of input values to the trusted system. Roughly speaking, within the trusted system, processes then compute the function F and synchronize before returning the result back into the untrusted system *[Benenson et al., 2006a]*. The main effort to achieve the security properties of SMC lies in solving the problem of uniform consensus in the trusted system.

Note that the property SMC-Validity differs slightly from the version considered in the original work on TrustedPals *[Benenson et al., 2006a]*. The original property required that F was computed on the inputs of all correct processes. Obviously, this property is impossible to achieve without reliable failure detection, making the change necessary.

7.2.3 Untrusted System: Assumptions

Within the untrusted system each pair of hosts is connected by a pair of unidirectional communication links, one in each direction. We assume reliable channels, i.e., every message inserted to the channel is eventually delivered at the destination. We assume no particular ordering relation on channels.

Timing Model

We assume that a local real-time clock is available to each process in the untrusted system, but clocks are not necessarily synchronized within the network. The untrusted system is assumed to be partially synchronous meaning that eventually, unknown bounds on all important network parameters (processing speed differences, message delivery delay) hold. The model is a variant of the partial synchrony model of Dwork, Lynch and Stockmeyer *[Dwork et al., 1988]*. The difference is that we assume reliable channels.

Failure model

The process failure model we assume in the untrusted system is the Byzantine failure model *[Lamport et al., 1982]*. A Byzantine process can behave arbitrarily. A process is correct if it follows its protocol. A process is *faulty* if it is not correct. We assume a majority of processes to be correct in the untrusted system.

7.2.4 Trusted System: Assumptions

The trusted system can be considered as an *overlay network* — a network that is built on top of another network — over the untrusted system. Nodes in the overlay network can be thought of as being connected by virtual or logical links. In practice, for example, smartcards could form the overlay network which runs on top of the Internet modeled by the untrusted processes. In the trusted system each process has an outgoing and an incoming reliable communication channel with every other process.

Trust Model

Within the trusted system we assume the existence of a *public key infrastructure*, which enables two communicating parties to establish confidentiality, message integrity and user authentication without having to exchange any additional secret information in advance. As mentioned above, we assume that the code running within the trusted system has been certified by some trusted authority, i.e., nodes in the trusted system may assume that each other's programs have not been tampered with. The trusted authority acts only during the setup phase of the system, not during the operational phase.

Timing Model

Security modules do not have any clock, they just have a simple step counter, whereby a step consists of possibly consuming a message, executing a local computation and possibly sending a message. Passing of time is checked by counting the number of steps executed. Roughly speaking, the timing assumptions for the processes in the trusted system are the same as those of the untrusted system, i.e., we assume partial synchrony. However, as we will explain next, in case the trusted process is associated with an untrusted process which is faulty, the trusted process may not rely on any timing assumptions whatsoever.

Failure Model

Like the untrusted system, the trusted system also is prone to attacks. However, the assumptions on the security modules and the possibility to establish secure channels reduce the options of the malicious hosts to attacks on the liveness of the system, i.e., (1) destruction of the security module, (2) interception of messages between the channel and the security module or (3) changes in the frequency of the step counter. This way, in the trusted system we assume the failure model of *general omission* and some asynchrony that can only affect to those trusted processes associated with faulty processes in the untrusted system (i.e., security modules residing on Byzantine hosts).

The general omission model considered in this chapter is the same as the one that we presented in Section 2.5.3 and considered in Chapter 6, that is, processes can fail by crashing or by omitting messages when sending or receiving. Permanent and transient omissions are possible.

Besides omissions, trusted processes can become arbitrarily slow (asynchronous) although the physical system in which they operate (i.e., the untrusted system) is partially synchronous. This models the effect of two different types of attacks by malicious hosts which we now explain: *timing attacks* and *buffering attacks*:

- Recall that security modules use a step counter to be aware of passing of time. The speed of the step counter is controlled by the associated host. In a *timing attack*, a malicious host arbitrarily changes the speed of the step counter of its security module. In that way, it can make the security module work faster (although the speed is physically bounded by the host's own clock) or slower (not bounded). As a consequence, the behaviour of its security module could become asynchronous and the communication through all the virtual channels which are adjacent to that security module would become asynchronous as well.
- As we have previously pointed out, a malicious host can intercept messages between its security module and the communication channels. Removal of intercepted messages is modeled as a message omission. In a *buffering attack*, the host does not remove the messages but stores them in a buffer and later on injects them into the communication channel after an arbitrary delay. This means that the message is not omitted but communication through that channel may become asynchronous in the trusted system.

Observe that both attacks affect the timing behaviour of the attacked security module. However, buffering attacks are more selective, since a particular communication channel of a security module could be attacked (become asynchronous in the trusted system) without affecting the rest of the communication channels of that security module.

In addition to the previous types of attacks, message reordering attacks are treated as a particular case of message buffering attacks, since in our algorithms no message is delivered if it is not the expected one (messages carry a unique sequence number). Also, note that the case of buffer overflow in the smartcard (e.g., if the attacker buffers a lot of messages and then passes all those messages at the same time to the smartcard) can be naturally treated as if the smartcard was faulty and omitted the reception of some message(s).

To summarize, processes in the trusted system can fail by crashing or omitting messages. Additional types of failure include the process or any of its incoming or outgoing communication channels becoming asynchronous. We call a process in the trusted system *correct* if it does not fail.

Connecting the Failure Assumptions

To connect the failure assumptions from trusted and untrusted systems we make the following assumption: a faulty security module implies a faulty host (i.e., a correct host implies a correct security module) but a faulty host does not necessarily imply a faulty

security module. This models the fact that attacks by a malicious host on its security module may not be successful.

The assumption implies that since there is a majority of correct processes in the untrusted system, there will be a majority of correct processes in the trusted system too.

7.2.5 Summary and Outlook

To summarize, the system model for TrustedPals is hybrid consisting of an untrusted system (representing the hosts) and an associated trusted system (representing the security modules). Timing is assumed to be partially synchronous for channels and processes in both models. The failure model in the untrusted system is Byzantine, while in the trusted system it is general omission with a subset of processes and channels becoming asynchronous.

In the remainder of this chapter we implement consensus in the *trusted* system. The original version of TrustedPals *[Benenson et al., 2006a]* assumed a *synchronous* setting for the trusted system. So the challenges lie in implementing consensus in a more asynchronous setting with omissions. We have chosen to follow the failure detector approach pioneered by *[Chandra and Toueg, 1996]*. In this approach, the synchrony assumptions are abstracted into a failure detector layer which offers information about failures of other processes. This allows us to implement consensus in an asynchronous manner on top of this layer.

Figure 7.3 summarizes the layers and interfaces of the proposed modular architecture for TrustedPals. A message exchange is performed in the transport layer, which is under control of the untrusted host. The security mechanisms for message encryption/decryption run in the TrustedPals layer on the security module. In the failure detector and consensus layers run the failure detection and consensus algorithms respectively. Finally, in the application layer, which again is under the control of the untrusted host, application software offering user interfaces to e-voting or SMC operate.

In the following section, we give a full formalization of the notions introduced above including the specific failure detector used. Then we take each layer at a time.

7.3 Integrating Failure Detection and Consensus Securely

Achieving security often relies on subtle issues not relevant to fault-tolerant synchronization. As an example, it is possible to successfully attack TrustedPals if the attacker can distinguish different message types on the network. Recall that the TrustedPals layer receives messages from the consensus protocol and from the failure detector and sends them over the network (see Figure 7.3). If the attacker does not forward all former messages while leaving the latter untouched, the failure detector would work properly but a consensus protocol would block forever. Therefore it is important to integrate the consensus and failure detector algorithms securely in TrustedPals.

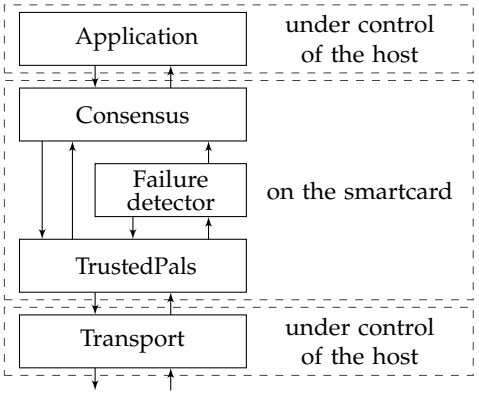


Figure 7.3: Architecture of the system with TrustedPals

We follow the integration mechanism proposed in the previous chapter (Section 6.7). In this way, we achieve that TrustedPals generates *fixed size* messages in *fixed* (periodic) time intervals.

7.3.1 Avoiding Message and Traffic Analysis

Traffic analysis refers to an attack technique which tries to derive information about messages by simply analysing the variation of the times in which they are sent and received. Since we send protocol messages "within" heartbeats we increase the difficulty for the attacker to distinguish an "empty" heartbeat from a "full" heartbeat by only looking at the timing of traffic. Since heartbeats are sent in an all-to-all pattern, it also hard to distinguish which process is sending protocol messages to which other process. This approach ensures *unobservability* regarding protocol messages, a notion known from the area of privacy-enhancing techniques *[Danezis and Diaz, 2008; Hansen and Pfitzmann, 2008]*.

Of course, the attacker could simply look into the contents of a heartbeat to discern a full from an empty message. That is why we employ cryptography on the channel. We implement a secure channel satisfying confidentiality, integrity and authenticity using standard techniques from cryptography *[Menezes, van Oorschot, and Vanstone, 1996]*. The idea is to encrypt and digitally sign all messages using the assumed public-key infrastructure and to use message authentication codes to ensure the integrity of the channel.

Important from a message-analysis point of view is that all heartbeats are indistinguishable from each other. As mentioned above, heartbeats are all the same length and if they are encrypted they will ideally look like random data. Hence, from just analysing the contents of a heartbeat it is impossible to distinguish a full from an empty heartbeat. Note that information about source and destination of a heartbeat must be sent unencrypted to allow routing. However, this information must also be stored within the encrypted part of the message to ensure authenticity.

7.3.2 Secure Scrambler

As in the previous chapter (Section 6.7), we assume a scrambler that allows to piggyback consensus messages into failure detector messages in the way explained in previous sections.

However, in this case cryptography is also applied to prevent and detect cheating and other malicious activities. We use a public key *cryptosystem* for encryption. Each message m in our model will be signed and then encrypted in order to reach authenticity, confidentiality, integrity, and non-repudiation.

The source and destination address are encrypted because this enables the receiver of a message to check whether the received message was intended for it or not and who the sender was. Thus, a malicious process cannot change the destination address in the header of a message from its security module and send it to an arbitrary destination without being detected. To detect a message deletion or loss, each message which is sent gets an identification number, where the fragment offset field determines the place of a particular fragment in the original message with same identification number.

7.4 Conclusions

In this chapter we can extract two different sets of conclusions.

- On the one hand, we have presented a modular redesign of TrustedPals, a smartcard-based security framework capable of efficiently solving Secure Multiparty Computation. The framework is based on a two-part architecture. One part represents the untrusted system, consisting of untrusted, Byzantine hosts. In the other part, representing the trusted system, security modules reduce the security problem to a fault-tolerant consensus among smartcards.

  The modular redesign allows TrustedPals to face the consensus problem in the general-omission failure model, which is more benign than the Byzantine model. Besides that, the trusted system has to deal with attacks which cannot be filtered by smarcards, specifically timing attacks and buffering attacks, resulting in a system model that includes asynchronous processes and/or channels.

  According to that, we use the eventually perfect failure detector class for the omission failure model defined in Chapter 6. We also propose to use the consensus algorithm presented in the same chapter.

  Another relevant aspect of the redesign is the integration of failure detection and consensus into the TrustedPals framework. Since the failure detector follows a heartbeat-based, all-to-all communication pattern, TrustedPals uses heartbeats as the transport mechanism for consensus messages. This approach, besides cryptography, ensures unobservability.

- On the other hand, we have shown a practical application of failure detectors in the general omission failure model. In this sense, we have proved that the failure

detector proposed in the previous chapter can be used as a building block to provide a solution in the hardest failure model (byzantine) by combining it with a security platform, TrustedPals.

Chapter 8

A Communication-Efficient Failure Detector in the General Omission Failure Model

In Chapter 5 we studied *communication efficiency* when implementing failure detectors in the crash failure model. In this chapter we address the definition and implementation of a communication-efficient failure detector for the general omission failure model, model which was previously considered in Chapter 6.

The approach we follow to define a communication-efficient failure detector class for the general omission failure model is very similar to the one we presented and followed in Chapter 6; we redefine the properties of the failure detector in terms of connectedness instead of correctness. In this sense, a *good* process[1] is a process that is able to (1) compute (i.e., they do not crash), and (2) communicate with a majority of processes in order to reach consensus. This implies that the communication ability of a process is represented by its connectivity degree, and is measured by the number of processes with which a process is able to communicate without omissions.

The key of the proposed failure detector implementation is that it relies on trusting relationships between pair of processes, represented by the *b-link* concept. Informally a b-link represents an abstraction of communication without failures between two processes, hence, an active b-link between two process implies that, till that moment, communication through the b-link has suffered no failure. Our failure detector looks for the minimal set of active b-links that allows to connect every process in a connected component of a graph. Additionally, a relaying mechanism allows all connected processes to share their information even if they are not directly connected. This way, our failure detector builds an overlay network of *correct enough* processes.

Finally, we also show how we could adapt the consensus algorithm of *[Chandra and Toueg, 1996]* to work with the proposed failure detector class.

Outline. In Section 8.1 we first present a brief introduction regarding communication efficiency in omission failure models. Section 8.2 defines the system model considered in this chapter. More precisely, the *b-link* abstraction is introduced in Section 8.2.1 and

[1] or *correct enough* or *well connected* process

presented as a basis for *well-connectedness* in Section 8.2.2. Failure detector properties are redefined for these concepts in Section 8.2.3, whereas communication efficiency for the general omission model is defined in Section 8.2.4. A communication-efficient implementation of a failure detector in the previous terms is shown in Section 8.3. Consensus integration for the previous failure detector class is explained in Section 8.4. Finally, Section 8.5 presents the conclusions of the chapter.

8.1 Introduction

Communication efficiency was already studied in Chapter 5 for the crash failure model. Additionally, it has already extended to some other scenarios, i.e. *[Martín and Larrea, 2010]* for the crash-recovery failure model or *[Arévalo, Jiménez, Larrea, and Mengual, To appear in 2011]*. In this case we apply this concept to the general omission failure model.

Previous works proposals on failure detection in omission failure models follow an all-to-all communication pattern (*[Delporte-Gallet et al., 2005 and 2007; Cortiñas et al., 2007]*) or the proposal in Chapter 6. That communication pattern involves a high communication cost in terms of sent messages. The failure detector we propose in this chapter follows an efficient communication pattern in the sense that eventually only a linear number of links will carry messages forever.

As in the case of the failure detector class presented in Chapter 6, we will redefine the properties of the failure detector class according to processes connectedness. In this sense, we look for *correct enough* or *well connected* processes, as *[Cortiñas et al., 2007; Delporte-Gallet et al., 2005 and 2007]* also do. The main difference is that we follow a communication pattern based on trust relationships between pairs of processes and that we provide the system with a mechanism to pause redundant communication in order to achieve communication efficiency, as mentioned above.

Although our failure detector achieves communication efficiency, there are some drawbacks that should be considered when comparing with previous works, for example, the failure detector implementation presented in Chapter 6. In this work, when failures occur, it takes the system a longer time to stabilize and work properly again. That higher reaction time is due to the fact that the system tries to keep communication cost low, whereas an all-to-all communication pattern provides a better reaction time. Additionally, the failure detector class from Chapter 6 considers *in-connected* and *out-connected* processes, whereas in this chapter we will only consider *connected* processes in a slightly more restrictive sense. As usually, we have to take into account a trade-off when improving a particular parameter, *communication efficiency* in this case.

8.2 System Model

As in the previous chapters, we model a distributed system as a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ in which every pair of processes is connected by a bidirectional communication link. Concerning timing assumptions, we consider a *partially synchronous* model in which there are bounds on relative process speed and message transmission times. Moreover, these bounds are not known and they hold only after some unknown but finite time, called GST (see Section 2.4.3). The communication links supporting this model are also called eventually timely links *[Aguilera, Delporte-Gallet, Fauconnier, and Toueg, 2008]*. However, we consider that communication links are reliable, i.e., every message put into a link is eventually received at the destination (although potentially omitted by the receiving process). We assume that every process has a local clock that can measure real-time intervals, although clocks are not synchronized.

We consider the general omission model, where processes can fail either by permanently crashing or by omitting messages. Omission failures can occur either while sending or while receiving messages, and can be transient, i.e., a process may temporarily omit messages and later on reliably deliver messages again. A process is *correct* if it does neither crash nor omit any message. Informally, we say that a process is *correct enough*, later on defined as *well-connected*, if it satisfies the following two conditions: (1) it does not crash, and (2) it is able to communicate in both directions and without omissions, either directly or indirectly, with a majority of processes. Observe that we are assuming the existence of a majority of *well-connected* processes.

8.2.1 Bidirectional Communication: *b-link*

We use the concept of *b-link* to represent the state of a bidirectional link. Given a pair of processes $(p, q) \in \Pi \times \Pi$, we denote by *b-link*$_{p \leftrightarrow q}$, equivalent to *b-link*$_{q \leftrightarrow p}$, the state of the bidirectional communication between processes p and q. At a given time, *b-link*$_{p \leftrightarrow q}$ can be in one of the following three possible states: *Active*, *Paused* and *Blocked*. When *b-link*$_{p \leftrightarrow q}$ is *Active*, p and q exchange messages periodically (in both directions). Instead, when *b-link*$_{p \leftrightarrow q}$ is *not Active*, i.e., it is either *Paused* or *Blocked*, p and q do not exchange messages periodically. An *Active b-link*$_{p \leftrightarrow q}$ becomes *Blocked* when either the communication between p and q is not timely or a message is omitted. An *Active b-link* that behaves timely and where the processes have not omitted any message can be paused in order to reduce the communication cost. Reciprocally, a *Paused b-link* can be activated in order to increase the process connectivity. Figure 8.1 shows the state diagram and state transitions for a *b-link*.

8.2.2 Well-Connectedness

Let $G = (V, E)$ be the undirected graph representing the system, where vertexes are processes, i.e., $V(G) = \Pi$, and edges are *Active b-link*s, i.e., $E = \{\{p, q\}$ such that *b-link*$_{p \leftrightarrow q}$ is *Active*$\}$. Due to crashes and omissions, G can be a disconnected graph with several connected components. All the processes belonging to a connected component

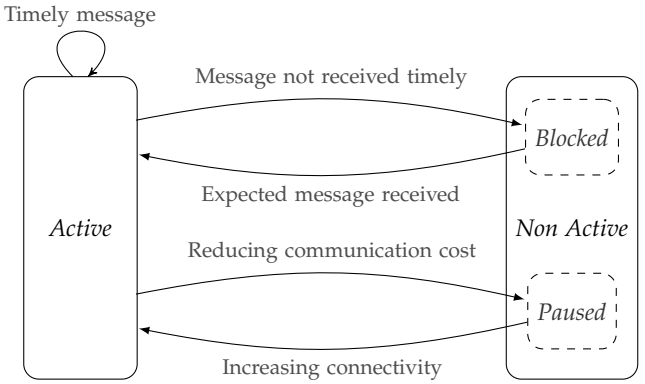


Figure 8.1: State diagram of a *b-link*

$S \subseteq G$ can communicate through a path of *Active b-link*s. Thus, we say that two processes $p, q \in V(S)$ are *connected*.

We assume in our model that every system contains a connected component S such that $|V(S)| \geq \lceil \frac{(n+1)}{2} \rceil$. We say that every process $p \in V(S)$ is *well-connected*.

8.2.3 Failure Detector Definition

The failure detector definition we propose for the general omission model is close to that of an Eventually Perfect failure detector for the crash model, denoted $\diamond\mathcal{P}$ and introduced by *[Chandra and Toueg, 1996]* (see Section 4.4). Indeed, we adapt the *correct/faulty* classification of processes in the crash model to the *well-connected/ not well-connected* classification in the general omission model. This failure detector satisfies the following completeness and accuracy properties:

- Strong Completeness. Eventually every *not well-connected* process is permanently considered as *not well-connected* by every *well-connected* process.
- Eventual Strong Accuracy. Eventually every *well-connected* process is permanently considered as *well-connected* by every *well-connected* process.

8.2.4 Communication Efficiency

We say that an algorithm is *communication-efficient* in the general omission model if it uses at most $n - 1$ bidirectional links to send messages forever. In the previously defined graph G, the minimum set of edges that connect all the vertexes, i.e., a spanning tree of G, has $n - 1$ edges. If G is a disconnected graph, in each connected component $S \subseteq G$ with m processes, a spanning tree of S will have $m - 1$ edges, so that in G there will be less than $n - 1$ edges.

8.3 Failure Detector Algorithm

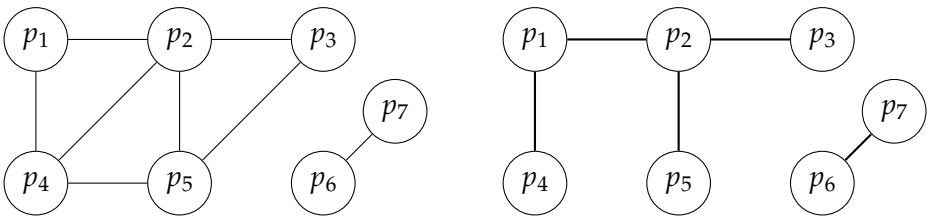
In this section, we present a communication-efficient failure detection algorithm satisfying the properties defined in the previous section. First we describe how the connectivity of processes is managed in order to get communication efficiency. Then, we get into details to explain how state transitions are implemented and formal properties satisfied.

8.3.1 Achieving Communication Efficiency

Our failure detection algorithm eventually uses at most $n - 1$ *Active b-link*s in order to maintain the n processes connected, and thus is communication-efficient. To achieve communication efficiency, every process p locally computes a spanning tree T for the connected component $S \subseteq G$ that p belongs to, using a deterministic version of the well known *breadth-first search* (BFS) Algorithm *[Knuth, 1997]*. The input to the BFS algorithm is the connectivity matrix of p, so every process in the same connected component eventually computes the same spanning tree. We assume that our BFS implementation starts with the process with the smallest identifier within each connected component and traverses the network in the order of increasing process identifier values.

Once a process p obtains the spanning tree T, p looks up which of its *Active b-link*s should be paused according to T, i.e., if a *b-link*$_{p\leftrightarrow q}$ is in S but not in T its state is set to *Paused*. The process in charge of pausing a *b-link*$_{p\leftrightarrow q}$ is always the process with the smallest identifier between p and q. This way, for each connected component an overlay network with $m - 1$ *Active b-link*s is built, being m the number of processes in the connected component.

Figure 8.2a shows an example of an undirected graph representing the connectivity of the system in a given execution. Figure 8.2b shows the result of applying the spanning tree algorithm to the graph. Paused links are not shown. Note that the following *b-link*s are paused: *b-link*$_{2\leftrightarrow 4}$, *b-link*$_{3\leftrightarrow 5}$ and *b-link*$_{4\leftrightarrow 5}$. According to the policy for pausing *b-link*s, process p_2 will pause *b-link*$_{2\leftrightarrow 4}$, process p_3 will pause *b-link*$_{3\leftrightarrow 5}$ and process p_4 will pause *b-link*$_{4\leftrightarrow 5}$.



(a) Connectivity graph **(b)** Spanning trees

Figure 8.2: Undirected graph of *b-link*s (a) and subsequent spanning trees (b)

8.3.2 The Algorithm

Algorithm 8.1, 8.2 and 8.3 present the proposed failure detection algorithm. For every process p, a set $connected_p$ stores p's perception of the connected component p belongs to, and henceforth provides the properties of the failure detector. In this regard, if $|connected_p| < \lceil \frac{(n+1)}{2} \rceil$, then process p does not consider itself as *well-connected*. On the other hand, if $|connected_p| \geq \lceil \frac{(n+1)}{2} \rceil$, then process p considers itself as *well-connected*, as well as all the processes in $connected_p$.

A *b-link*$_{p \leftrightarrow q}$ is implemented by a pair of variables, $link_p[q].state$ at process p, and $link_q[p].state$ at process q. For the sake of brevity, in the rest of this section we will use $link_p[q]$ to refer to $link_p[q].state$.

A Boolean matrix M_p is used by every process p to represent the connectivity of the whole system. M_p represents the adjacency matrix of an undirected graph where $M_p[p][q]$ shows if there is an edge between p and q. If $M_p[p][q] = M_p[q][p] = 1$ then *b-link*$_{p \leftrightarrow q}$ is *Active*, while if $M_p[p][q] = M_p[q][p] = 0$ then *b-link*$_{p \leftrightarrow q}$ is non-*Active* (either *Paused* or *Blocked*). Figure 8.3 shows the matrices representing the graphs of Figure 8.2.

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{pmatrix}$$

(a) For graph in Fig. 8.2a **(b)** For graph in Fig. 8.2b

Figure 8.3: Connectivity matrices M_p for graphs of Figure 8.2

The algorithm is based on the periodical exchange of heartbeat messages. Every message sent by p carries M_p as well as a version number $V_p[q]$ for each row q of the matrix. A process p updates M_p whenever a *b-link*$_{p \leftrightarrow q}$, for any q, changes. Also, p updates M_p from the matrix M_q carried by every message received from q (in this case, vectors V_p and V_q are compared in order to get the most up-to-date information regarding the connectivity of the system).

We now describe the four main tasks that the algorithm executes. In Task 1, every process p periodically sends an ALIVE message through each one of its *Active b-link*s. In order to detect message omissions, every message that p puts into a link has an associated sequence number, identified by $link_p[q].send\text{-}seq$.

In Task 2, every process p waits for the next ALIVE message from each one of its *Active b-link*s. If p does not receive the next expected message from a process q timely, then $link_p[q]$ is changed from ACTIVE to BLOCKED and the corresponding timeout value is incremented. When $link_p[q]$ changes to BLOCKED, p stops sending ALIVE messages to q, and therefore $link_q[p]$ will eventually change to BLOCKED too. Observe that if the timeout has expired due to a message omission, the *b-link* will remain *Blocked*

Algorithm 8.1: Comm.-efficient FD in Omission: main algorithm

```
1  Procedure main()
2      connected_p ← Π
3      foreach q ∈ Π do
4          link_p[q].state ← ACTIVE ; link_p[q].buffer ← ∅
5          link_p[q].send-seq ← 1 / link_p[q].recv-seq ← 1          {msg.-id to be sent to / received from q}
6          link_p[q].timeout ← default time-out interval
7          foreach u ∈ Π do M_p[q][u] ← 1
8          V_p[q] ← 0                                               {version number for every row of M_p}

9  || Task 1: repeat periodically                                   {Sending heartbeats}
10     foreach q ∈ Π − {p} do
11         if link_p[q].state = ACTIVE then send-message(ALIVE, link_p[q].send-seq++, M_p, V_p) to q

12 || Task 2: repeat periodically                                   {Checking time-outs}
13     if ( link_p[q].state = ACTIVE and
            p has not received (ALIVE, link_p[q].recv-seq, M_q, V_q) from q ≠ p
            during the last link_p[q].timeout time units of p's clock ) then
14         link_p[q].timeout++
15         change-link-state(q, BLOCKED)

16 || Task 3: when receive a message m (type, id, M_q, V_q) from q   {Processing msgs in order}
17     insert m into link_p[q].buffer
18     while ( ∃ a message m' with (type, id, M_q, V_q) in
               link_p[q].buffer such that id = link_p[q].recv-seq ) do
19         if link_p[q].state = BLOCKED then change-link-state(q, ACTIVE)
20         if type = START and link_p[q].state = PAUSED then change-link-state(q, ACTIVE)
21         if type = PAUSE and link_p[q].state = ACTIVE then change-link-state(q, PAUSED)
22         foreach u ∈ Π − {p} do
23             if V_q[u] > V_p[u] then
24                 foreach v ∈ Π do M_p[u][v] ← M_q[u][v]
25                 V_p[u] ← V_q[u]
26         remove m' from link_p[q].buffer
27         link_p[q].recv-seq++

28 || Task 4: when M_p changes do                                   {Check connectivity}
29     connected_p ← calculate-set-of-connected-processes(M_p)
30     if |connected_p| < ⌈(n+1)/2⌉ then
31         q ← process r with the smallest id with (link_p[r].state = PAUSED) and (r ∉ connected_p)
32         if q ≠ null then
33             change-link-state(q, ACTIVE)
34             send-message(START, link_p[q].send-seq++, M_p, V_p) to q
35     candidates_p ← get-redundant-b-links(M_p)
36     foreach q ∈ candidates_p do
37         change-link-state(q, PAUSED)
38         send-message(PAUSE, link_p[q].send-seq++, M_p, V_p) to q
```

Algorithm 8.2: Comm.-efficient FD in Omission: *change-link-state* procedure

```
39 Procedure change-link-state(q:process-id; newState:state-type)
40     link_p[q].state ← newState
41     M_p[p][q] ← (newState = ACTIVE)
42     V_p[p]++
```

Algorithm 8.3: Comm.-efficient FD in Omission: *get-redundant-b-links* procedure

Result: Obtains a spanning tree

```
43 Procedure get-redundant-b-links()
44   V ← connected_p
45   A ← ∅
46   P ← ∅
47   P ← P+ min-ident(connected_p)
48   V ← V− min-ident(connected_p)
49   while V ≠ ∅ do
50     u ← first-process-in(P)
51     P ← P − u
52     foreach r in ∈ V, ordered by identif. do
53       if M_p[u][r] = 1 and M_p[r][u] = 1 then
54         P ← P + r
55         V ← V − r
56         A ← A + {u, r}
57   Cand ← ∅
58   foreach q ∈ Connected_p being p < q do
59     if {p, q} ∉ A and link_p[q].state = ACTIVE then Cand ← Cand + q
60   return Cand
```

forever. Otherwise, i.e., if the timeout was premature, the increment of the timeout value guarantees that eventually the expected message, unless omitted, will always be received timely.

In Task 3, received messages are delivered following their sequence number. If the message is received through a BLOCKED $link_p[q]$, it consequently becomes ACTIVE, and by Task 1 process p starts sending ALIVE messages to q again. The *b-link* state transitions between *Active* and *Paused* of Figure 8.1 are implemented too, and some rows of matrix M_q are copied into M_p if needed.

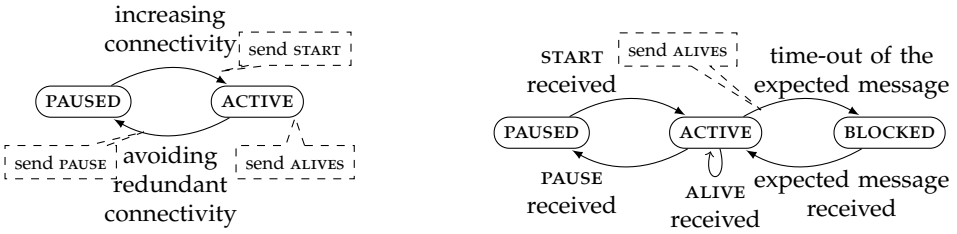
Finally, Task 4 is executed by a process p whenever M_p changes (either in Task 2 or in Task 3) in order to update $connected_p$. Furthermore, Task 4 adjusts p's connectivity if necessary. In case p needs to increase its connectivity, it will send START messages to processes with which it has a *Paused b-link*. This task also checks if there are redundant *Active b-link*s (by the *get-redundant-b-links* procedure of Algorithm 8.3, which implements the previously presented BFS algorithm).

To pause an *Active b-link*$_{p \leftrightarrow q}$, process p sends a PAUSE message to q.

Figure 8.4 shows a detailed state diagram according to link states. More precisely, Figure 8.4a, shows the state transitions implemented in Task 4, whereas Figure 8.4b depicts actions in Tasks 2 and 3.

8.3.3 Correctness Proof

We now show that the presented algorithm satisfies the failure detector properties defined in Section 8.2. Furthermore, we also show that the algorithm is communication-efficient.



(a) State diagram of a *b-link* for a process when adjusting its connectivity

(b) State diagram of a *b-link* for a process that does not need to change its connectivity

Figure 8.4: Detailed diagram of states of a *b-link*.

The proof is divided in three parts. First, Lemmas 8.1 to 8.5 show that, for every two processes p and q, the variables $link_p[q]$ and $link_q[p]$ represent consistently the behaviour defined for *b-link*$_{p \leftrightarrow q}$. Then, Lemmas 8.6 to 8.8 show that the system eventually converges, i.e., eventually all the processes in each connected component agree permanently on the same set of *Active b-link*s. Finally, Lemmas 8.9 and 8.10 show that the $connected_p$ variable at each *well-connected* process p satisfies the completeness and accuracy properties defined in Section 8.2.

Lemma 8.1: *If an* ACTIVE $link_p[q]$ *is set to* BLOCKED, *then eventually either* $link_q[p]$ *is set to* BLOCKED *or* PAUSED, *or* $link_p[q]$ *will not remain* BLOCKED.

Proof. By Task 2, a process p that does not receive the next expected message from q ($link_p[q]$.*recv-seq*) in time by an *Active b-link*$_{p \leftrightarrow q}$ sets $link_p[q]$ to BLOCKED (Line 18). If the next expected message from q has been omitted, $link_p[q]$ is set to BLOCKED permanently. As a consequence, p stops sending ALIVE messages to q (by Task 1 of p) permanently, and if $link_q[p]$ is ACTIVE it will be set to BLOCKED permanently too (Line 18). If the next expected message from q has not been omitted, this message will eventually arrive to p and $link_p[q]$ will be set to ACTIVE again (Lines 22-23). If when p receives the expected message from q, sets $link_p[q]$ to ACTIVE again and sends by Task 1 an ALIVE message to q which is received in q before its timeout on p triggers, then $link_q[p]$ will remain ACTIVE. Otherwise, if the timer on p triggers at q, $link_q[p]$ will be set to BLOCKED temporarily. Finally, note that if q concurrently has set $link_q[p]$ to PAUSED, $link_q[p]$ will remain PAUSED, since no messages from p are waited by Task 2 of q if $link_q[p]$ is PAUSED, and thus the transition to BLOCKED does not apply. □

Lemma 8.2: *If an* ACTIVE $link_p[q]$ *is set to* PAUSED, *then eventually* $link_q[p]$ *is set to* PAUSED *or* BLOCKED.

Proof. By the procedure *get-redundant-b-links* (Algorithm 8.3), a process p tries to pause *b-link*$_{p \leftrightarrow q}$ by sending a PAUSE message to q and changing $link_p[q]$ to PAUSED (Lines 41-44). Observe that, according to our policy for pausing *b-link*s, an *Active b-link*$_{p \leftrightarrow q}$ is paused only by the process with the smallest identifier between p and q.

When a $link_p[q]$ is set to PAUSED, if the PAUSE message sent by p is not eventually received in q, by Task 2, q sets $link_q[p]$ to BLOCKED permanently. If the PAUSE message sent by p is eventually received in q and $link_q[p]$ is ACTIVE, by Task 3, q will set $link_q[p]$ to PAUSED (Lines 26-27). Otherwise, if $link_q[p]$ is BLOCKED due to a premature timeout, $link_q[p]$ will be set first to ACTIVE (Lines 22-23 of Task 3 of q) and then to PAUSED (Lines 26-27). □

Lemma 8.3: *If a* BLOCKED $link_p[q]$ *is set to* ACTIVE*, then eventually either* $link_q[p]$ *is set to* ACTIVE*, or* $link_p[q]$ *will not remain* ACTIVE*.*

Proof. By Task 3 (Line 23) a process p sets a BLOCKED $link_p[q]$ to ACTIVE when the next expected message, $link_p[q].recv\text{-}seq$, is received. Consequently, by Task 1, p starts to send ALIVE messages to q. If $link_q[p]$ is BLOCKED and q eventually receives the next expected message from p, q will set $link_q[p]$ to ACTIVE and, by Task 1, q will send ALIVE messages to p. Otherwise, if the next expected message from p to q is omitted, by the reasoning followed in Lemma 8.1, $link_q[p]$ is set to BLOCKED permanently, and hence $link_p[q]$ will not remain ACTIVE. Finally, note that if $link_q[p]$ is set to PAUSED concurrently, by Lemma 8.2 $link_p[q]$ is set to PAUSED or BLOCKED, i.e., it will not remain ACTIVE. □

Lemma 8.4: *If a* PAUSED $link_p[q]$ *is set to* ACTIVE*, then eventually either* $link_q[p]$ *is set to* ACTIVE*, or* $link_p[q]$ *will not remain* ACTIVE*.*

Proof. By Task 4 (Lines 36-40), a process p tries to activate $b\text{-}link_{p \leftrightarrow q}$ (in order to increase its connectivity) by sending a START message to q and changing $link_p[q]$ to ACTIVE. If $link_q[p]$ is PAUSED and q receives the START message, by Task 3, q sets $link_q[p]$ to ACTIVE (Lines 24-25) and consequently, by Task 1, q starts to send ALIVE messages to p. Otherwise, if q does not receive the START message from p, $link_q[p]$ will not be set to ACTIVE. Thus, q will not start to send ALIVE messages to p and therefore, $link_p[q]$ will be set to BLOCKED. Finally, note that if $link_q[p]$ is permanently BLOCKED (due to an omission from p to q), then by the reasoning followed in Lemma 8.1, $link_p[q]$ will be set to BLOCKED too. □

Lemma 8.5: *The communication between every pair of processes* $p, q \in \Pi$ *corresponds to the behaviour defined for* b-link*s.*

Proof. By Lemmas 8.1, 8.2, 8.3 and 8.4 the communication from p to q, determined by the variable $link_p[q]$, and the communication from q to p, determined by the variable $link_q[p]$, will eventually have a consistent behaviour, which is the one defined for the corresponding $b\text{-}link_{p \leftrightarrow q}$. Observe that the variables $link_p[q]$ and $link_q[p]$ will not have permanently the combination of values (ACTIVE, non-ACTIVE), where non-ACTIVE is either BLOCKED or PAUSED. □

Lemma 8.6: *Eventually,* b-link*s will block only due to omissions and they will be blocked permanently.*

Proof. By Lemma 8.1, a *b-link*$_{p\leftrightarrow q}$ switches to the state *Blocked* when process p (or q) does not receive an expected message in time. This may occur either because the timer *link*$_p[q]$.*timeout* triggered (in Task 2 of p) before the message was received, or because the message was omitted. Since the system is partially synchronous, and since, by Task 2 (Line 17), the timeout associated with the *link*$_p[q]$ variable (*link*$_p[q]$.*timeout*) is incremented whenever the link state is set to BLOCKED, eventually no expected message will be received after the timer triggers. Thus, eventually, the only reason for any *b-link* to be *Blocked* will be a message omission, and henceforth the *b-link* will be *Blocked* permanently. □

Observation 8.1: *At every process p, the matrix M_p is updated with its own connectivity information and with the matrices M_q received in the* ALIVE *messages. Every time p changes matrix M_p in the procedure* change-link-state *(Lines 45-48) the version number $V_p[p]$ is incremented. The updated M_p and its version number V_p are sent with p's next* ALIVE *message at least to one process in the same connected component. Observe that when p delivers a message in Task 3, M_p is updated with M_q comparing the version numbers V_p and V_q, and therefore, copying only the last version of the connectivity information (Lines 28-31). Besides, the local delay in process p for relaying M_p and V_p, δ, is bounded in the algorithm by the period of Task 1 of p, which is finite if p is eventually synchronous and has not crashed. This way, implicitly, a relaying mechanism of the last version of M_p and V_p is obtained among the processes in the same connected component.*

Lemma 8.7: *Eventually every process p will not activate more* b-link*s.*

Proof. By the definition of connected component and by Lemma 8.6, eventually for every two processes p and q that belong to different connected components, *b-link*$_{p\leftrightarrow q}$ will be *Blocked* permanently. Observe that eventually, by Observation 8.1, all the processes in the same connected component will share the same matrix M_p with the connectivity information. In Task 4 (Lines 36-40), processes activate *b-link*s trying to connect to a majority of processes. If a process p is *well-connected*, it will eventually be in a connected component with at least $\lceil \frac{(n+1)}{2} \rceil$ processes. After that, p will not activate more *b-link*s (condition in Line 36). If a process q is not a *well-connected* process, by definition, it will belong to a connected component with less than $\lceil \frac{(n+1)}{2} \rceil$ processes. However, eventually q will not have any *Paused b-link* with processes outside its connected component, so q will not be able to activate more *b-link*s (Lines 37-38). □

Lemma 8.8: *Processes in the same connected component will calculate eventually and permanently the same set of Active* b-link*s (eventually the state of a* b-link *does not change from* Active *to* Paused *or vice versa).*

Proof. From Lemma 8.5, the communication between two processes p, q is always bidirectional, in the sense that both processes exhibit a consistent behaviour regarding *b-link*$_{p\leftrightarrow q}$. According to Observation 8.1, when p activates a *b-link*, all the processes in the same connected component will receive this information and will have the same connectivity matrix, M_p; therefore they will share the same set of *Active b-link*s. Observe

that the algorithm used to pause *b-link*s in Task 4, procedure *get-redundant-b-links*, is deterministic. Henceforth, all the processes in the same connected component will calculate the same set of *Active b-link*s. Since by Lemma 8.7 eventually no *b-link* will be activated, and consequently no more *b-link*s will be paused, the set of *Active b-link*s will be permanently identical for every process p that belongs to the same connected component. □

Lemma 8.9: *Eventually and permanently, for every* well-connected *process p every* not well-connected *process* $q \notin connected_p$.

Proof. If q is a *not well-connected* process, by Lemma 8.8 and by the definition of *not well-connected* process, eventually q is permanently in a connected component with less than $\lceil \frac{(n+1)}{2} \rceil$ processes. Being p a *well-connected* process, by definition, p is in a connected component with at least $\lceil \frac{(n+1)}{2} \rceil$ processes. Therefore p and q will belong to different connected components, and by the algorithm and Lemma 8.8, $q \notin connected_p$ permanently. □

Lemma 8.10: *Eventually and permanently, for every* well-connected *process p every* well-connected *process* $q \in connected_p$.

Proof. If q and p are *well-connected* processes, they will activate *b-link*s to connect to a majority of processes, and by Lemma 8.8, eventually they will calculate permanently the same set of *Active b-link*s and they will be connected with a majority of the processes permanently. If both p and q are connected to a majority of the processes they must be in the same connected component, and by the algorithm and Lemma 8.8, $q \in connected_p$ permanently. □

Theorem 8.1: *The algorithm of Figure 8.1 implements the properties of Strong Completeness and Eventual Strong Accuracy defined in Section 8.2.*

Proof. Directly from Lemmas 8.9 and 8.10. □

Theorem 8.2: *Eventually the number of* b-link*s used in the system is permanently* $n - 1$ *or lower.*

Proof. By Lemma 8.8, eventually every connected component will not change, and *Active b-link*s will express the connectivity in the connected component, i.e., the number of *b-link*s used permanently. The procedure *get-redundant-b-links* executes a BFS algorithm that guarantees a spanning tree of *Active b-link*s in a connected component. By the definition of connected component and by Lemma 8.6, eventually every $b\text{-}link_{p \leftrightarrow q}$ for p and q in different connected components will be *Blocked* forever. Therefore, since the spanning tree of a graph with k nodes has $k - 1$ edges, the number of *Active b-link*s in the system will be at most $n - 1$. □

8.4 Consensus

We describe now how the proposed failure detector can be used to solve consensus. Remember that in the consensus problem, every process proposes a value, and correct processes must eventually decide on some common value that has been proposed. In Section 6.6.1 we proposed a redefinition of the termination property of consensus in order to adapt it to the general omission model. We also follow that approach in this chapter, but adapting it to the *well-connected* concept (instead of considering *in/out-connectivity*).

Termination: Every *well-connected* process eventually decides some value.

The rest of properties, namely integrity, agreement and validity, do not need to be redefined.

Along the same line as Chapter 6, instead of designing a new consensus algorithm from scratch we propose an adaptation of the well-known consensus algorithm of *[Chandra and Toueg, 1996]* for crash-prone systems (included in Appendix A). The adaptation is close to the one proposed in Section 6.6.2, but differs in what concerns the use of the overlay network provided by the underlying failure detector. Process connectivity, and specifically the component containing *well-connected* processes, provides the necessary relaying framework for consensus messages to be delivered. Therefore, consensus messages follow a path of *Active b-link*s to reach their destination. Proceeding in this way, the communication cost of the consensus algorithm remains linear too. Nevertheless, contrary to the proposal of Chapter 6, where the underlying failure detector uses a permanent all-to-all communication pattern which eases the adaptation, the fact that our failure detector is communication-efficient forces the consensus layer to check if the overlay network has changed during the execution of the current round of the algorithm, in which case the round is aborted in order to avoid blocking.

Observe that consensus messages should also be monitored in order to detect omissions, so we propose to integrate consensus messages and failure detector messages as proposed in Section 6.7.

8.5 Conclusions

In this chapter we have presented a communication-efficient failure detector for the general omission model. We give a new definition of the $\diamond\mathcal{P}$ failure detector, in terms of its completeness and accuracy properties, based on process connectivity rather than on process correctness. The connectivity is obtained by means of bidirectional communication between processes, based on the b-link concept. Besides, we provide an implementation of the failure detector that is communication-efficient. The presented algorithm uses only $n-1$ bidirectional links carrying messages forever. We have also discussed how to use this failure detector to solve consensus and shown that consensus algorithm by *[Chandra and Toueg, 1996]* can be easily adapted to use this failure detector.

Comparing with the proposal of Chapter 6, we should point out the following differences:

- Correct enough concept. In Chapter 6, a process can be in-connected or out-connected (or even both at once). However, in this chapter correct enough processes have to be connected at input and at output with the same process(es). The latter approach is more restrictive than the former.
- Communication pattern. In the previous proposal, the communication pattern was based on a periodic all-to-all message exchange, which is more suitable for masking communication between processes and provides a better reaction time when failures occur. In contrast, the communication-efficient failure detector tries to use a minimal number of links to communicate, which offers hints to a malevolent adversary about system state and how to capture information. Additionally, a failure may imply that a set of processes gets disconnected from the main set temporarily (until new b-links are activated).
- System assumptions. The system presented in Chapter 6 allowed a minority of processes to behave asynchronously. In this chapter we have assumed that all components are eventually timely.

As a conclusion, despite the advantages of the failure detector proposed in this chapter, it is not suitable for being used as a building block to solve SMC as presented in Chapter 7.

As a final remark we would like to mention an interesting open question regarding the implementation of the failure detector in this chapter. Every process in the system has the possibility of activating and pausing b-links, depending on their connectivity. We have proposed to activate and pause links according to their identifiers. However, another criteria could be used, such as achieving a given topology or taking into account the number of suspicions (in order to eventually avoid asynchronous channels/processes).

Chapter 9

Overall Conclusions and Future Work

In this work we have presented some interesting results regarding failure detection in asynchronous systems.

First, in the context of the classical crash failure model, we have considered communication efficiency of failure detector implementations. We have shown that it can be extended by defining communication optimality. More precisely, we have proved that c is the minimal number of links carrying messages forever that are needed to implement failure detectors in the crash failure model, being c the number of correct processes.

We have also shown that failure detectors can also be implemented in failure models beyond the crash failure mode. In this sense, instead of considering correct or faulty processes, we consider *correct enough*, also called *good*, processes. In the case of the general omission failure model, we propose to consider correct enough processes in terms of connectedness, i.e., a *good* process has to be able to compute and to communicate with at least one correct process. Following this path, we have defined a new failure detector class for the general omission failure model and shown an implementation. We have also explained how the classical consensus algorithm of Chandra and Toueg can be easily adapted to work with failure detectors belonging to that class.

Another important advantage of failure detectors, its modularity, has been emphasized by using a failure detector for the general omission failure model as a building block to provide a solution for a weaker system (which considers the Byzantine failure model), by combining it with a secure platform called TrustedPals. In this sense, we also present a proposal for solving Secure Multiparty Computation, a problem in the security researching area, in asynchronous systems.

Following one of our main goals, i.e., achieving communication efficiency, we have also presented a communication-efficient implementation of failure detectors for the general omission failure model. In this case we have followed an interesting communication pattern, based on the *b-link* abstraction, that allows to build a functional overlay network in environments where omissions may happen.

The following list summarizes the most remarkable contributions presented in this dissertation:

- Definition of communication optimality for the crash failure model.
- Communication-optimal implementations of $\diamond\mathcal{P}$ (and $\diamond\mathcal{Q}$) failure detectors for the crash model.
- Introduction and issues regarding failure detection in the general omission failure model.
- Definition and implementation of a $\diamond\mathcal{P}$ failure detector class for the general omission failure model. We have also included an adaptation of the classical Chandra and Toueg's consensus algorithm to the new defined failure detector class.
- Example of use, in which we show how to solve Secure Multiparty Computation in an asynchronous system with the Byzantine failure model by transforming it into solving the consensus problem in the general omission failure model with some additional asynchrony.

  There are some interesting contributions embedded in this subject:

  - We introduce Secure Multiparty Computation, an interesting security problem, and TrustedPals, a framework to solve the previous problem in synchronous systems. Additionally, we show that failure detectors can be used as a building block to provide a solution to Secure Multiparty Computation in asynchronous systems as well.
  - We directly use the new failure detector class for the general omission failure model previously defined in this work to solve a different problem in a different failure model. We consider that this is a wonderful justification of the general omission failure model and of our defined failure detector class for such model.
  - It is worth noting that failure detectors, which have traditionally been blamed for being too theoretical, can be used to solve a more practical problem.
- Definition and implementation of a communication-efficient failure detector for the general omission failure model. In addition to the communication-efficient implementation, the *b-link* abstraction is presented. This communication pattern approach provides a flexible way to build overlay networks with a high ability to adapt to different scenarios, as presented in the following section.

Future work

There are some open lines that could be explored beyond this work.

Regarding communication optimality, there are many other measures that we should look into in order to be able to consider the trade-offs when implementing failure detectors in different scenarios. Besides, we have defined communication optimality for the crash failure model, but not for the omission failure model, so the following

question arises; is it possible to define and achieve communication optimality in the general omission failure model?

On the other hand, the system models proposed in this work for implementing failure detectors are rather strong in terms of required synchrony. The system model considered in Chapter 6 allows some processes to behave asynchronously, but in the rest of the cases, processes and communication channels are assumed to be partially synchronous.

Additionally, it would be interesting to adapt the communication-efficient implementation of Chapter 8 to a weaker system model, in which, for example, some channels are asynchronous or fair-lossy. As stated in the conclusions of that chapter, we could set priorities when choosing channels and hence achieve some given overlay topology or dynamic adaptation to the underlying network.

Another interesting point could be determining the weakest failure detector in different failure models for other problems such as k-set agreement.

Regarding our proposal to solve SMC in the asynchronous system presented in Chapter 7, we also intend to implement that approach and perform practical experiments.

Appendices

Chapter A

Solving Consensus with a $\diamond\mathcal{S}$ Failure Detector

[Chandra and Toueg, 1996] showed how to solve consensus using a failure detector $\mathcal{D} \in \diamond\mathcal{S}$ through Algorithms A.1 and A.2. Since $\diamond\mathcal{P}$ is stronger than $\diamond\mathcal{S}$, such algorithm is also valid for failure detectors of class $\diamond\mathcal{P}$.

System model. As presented in Section 2.2, a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ is assumed. Processes may crash, but there is a majority of correct processes. Communication channels between correct processes are reliable. The system is asynchronous.

Description. The algorithm follows a rotating coordinator model (Algorithm A.1). There are several rounds and one coordinator per round; each time the current coordinator is suspected to have crashed, then suspecting processes move on to the following round. However, the accuracy property of the $\diamond\mathcal{P}$ failure detector class states that eventually there is a time after which correct processes are not suspected by any correct process. Hence, there is a time after which the coordinator will not be suspected by a majority of processes, which implies that it will be able to lead to consensus.

Note that Reliable Broadcast primitives (see Appendix B) are used when broadcasting and delivering the decision (Phase 4, lines 36 and 37).

A.1 Example of Execution

Figure A.1 shows an example of an execution of the consensus algorithm.

Observe that the decision can be a value proposed by a non-correct process, e.g., p_5. It is worth noting that even faulty processes are able to decide if they crash after receiving the decision (Phase 4).

Algorithm A.1: Solving consensus in the crash model using a $\mathcal{D} \in \diamond\mathcal{S}$: managing proposals

```
{Every process p executes the following}
1  Procedure propose(v_p)
2      estimate_p ← v_p                                   {estimate_p is p's estimate of the decision value}
3      state_p ← undecided
4      r_p ← 0                                            {r_p is p's current round number}
5      ts_p ← 0                                           {ts_p is the last round in which p updated estimate_p, initially 0}
       {Rotate through coordinators until decision is reached}
6      while state_p = undecided do
7          r_p ← r_p + 1
8          c_p ← (r_p mod n) + 1                          {c_p is the current coordinator}
9          Phase 1: {All processes p send estimate_p to the current coordinator}
10             send (p, r_p, estimate_p, ts_p) to c_p
11         Phase 2: { The current coordinator tries to gather ⌈(n+1)/2⌉ estimates. If it succeeds,
                      it proposes a new estimate. Otherwise, it sends a NEXT message to all }
12             if p = c_p then
13                 wait until for ⌈(n+1)/2⌉ processes q: received (q, r_p, estimate_q, ts_q) from q
14                 msgs_p[r_p] ← {(q, r_p, estimate_q, ts_q) | p received (q, r_p, estimate_q, ts_q) from q}
15                 t ← largest ts_q such that (q, r_p, estimate_q, ts_q) ∈ msgs_p[r_p]
16                 estimate_p ← select one estimate_q such that (q, r_p, estimate_q, t) ∈ msgs_p[r_p]
17                 send (p, r_p, estimate_p) to all
18         Phase 3: {All processes wait for the new estimate proposed by the coordinator}
19             wait until received [ (c_p, r_p, estimate_{c_p}) or (c_p, r_p, NEXT)] from c_p or c_p ∈ D
20             if received (c_p, r_p, estimate_{c_p}) from c_p then
21                 estimate_p ← estimate_{c_p}
22                 ts_p ← r_p
23                 send (p, r_p, ack) to c_p
24             else
25                 send (p, r_p, nack) to c_p
26         Phase 4: { If the current coordinator sent a valid estimate in Phase 2, it waits for replies of processes.
                      If ⌈(n+1)/2⌉ processes replied with ack, the coordinator R-broadcasts a decide message }
27             if p = c_p then
28                 wait until for all process q: ( received (q, r_p, ack) or
                                                   received (q, r_p, nack) )
29                 if for ⌈(n+1)/2⌉ processes q:received (q, r_p, ack) then
30                     R-broadcast(p, r_p, estimate_p, decide)
```

Algorithm A.2: Solving consensus in the crash model using a $\mathcal{D} \in \diamond\mathcal{S}$: adopting the decision

```
{If p R-delivers a decide message, p decides accordingly}
31 when R-deliver(q, r_q, estimate_q, decide) do
32     if state_p = undecided then
33         decide(estimate_q)
34         state_p ← decided
```

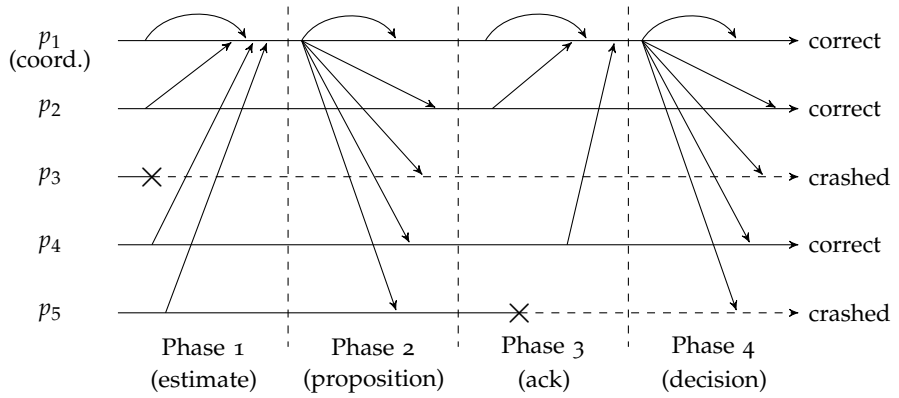


Figure A.1: Example of an execution of the consensus algorithm

Chapter B

Reliable Broadcast

Reliable Broadcast *[Hadzilacos and Toueg, 1994]* (earlier version in *[Hadzilacos and Toueg, 1993]*) is a communication primitive that allows to disseminate (by broadcasting) information among a set of processes in a reliable way.

Roughly speaking, if a correct process sends a message to a set of processes, every correct process in that set should deliver that message. Although it could seem a simple task, it is not that easy if failures, such as crashes or message losses, can occur. To set an example, if a process p sends a message m to a set of processes and at least one of them does not receive m, retransmission could not be enough since p could crash before being able to send it again.

Formally, Reliable Broadcast is defined in terms of two primitives, namely *R-broadcast(m)* and *R-deliver(m)*, and there are several Reliable Broadcast problem specifications, depending on the properties they provide. In our case we use *Regular Reliable Broadcast [Guerraoui and Rodrigues, 2006]*, which provides the following properties *[Hadzilacos and Toueg, 1994]*:

- RB-Validity. If a correct process R-broadcasts a message m, then it eventually R-delivers m.
- RB-Agreement. If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.
- RB-Uniform integrity. For any message m, every process R-delivers m at most once, and only if m was previously R-broadcast by $sender(m)$.[1]

Observe that *RB-Uniform integrity* property includes the *no duplication* property. Also note that we do not consider that channels are able to create messages (also called *no creation* property).

[1] We assume that messages include the identity of the sender and a sequence number, which make every message unique.

Reliable Broadcast guarantees that all correct processes deliver the same set of messages. This set includes at least all messages broadcast by correct processes. Algorithm B.1, already included in *[Chandra and Toueg, 1996]*, presents a simple Reliable Broadcast algorithm for asynchronous systems with up to $n - 1$ crash failures and reliable channels. Informally, when a process receives a message for the first time, it relays the message to all processes and then R-delivers it.[2]

Algorithm B.1: Reliable Broadcast by message diffusion

{Every process p executes the following}

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

3  R-deliver(m) occurs as follows:
4      when receive m for the first time do
5          if sender(m) ≠ p then send m to all
6          R-deliver(m)
```

Reliable Broadcast is used in some algorithms in this work. One of the proposed communication-optimal failure detectors in the crash failure model uses its primitives (Section 5.5.1). Reliable Broadcast is also used in the consensus algorithm of *[Chandra and Toueg, 1996]* for solving consensus using a $\diamond\mathcal{S}$ failure detector class and thus in our adaptations of this algorithm to work with the proposed failure detectors.

[2] An optimization consists in not relaying m to p, $sender(m)$ and the process q from which m has been received for the first time (if $q \neq sender(m)$).

Chapter C

$\diamond\mathcal{P}$ Implementations Referenced in Chapter 5

In this appendix we include two implementations of the eventually perfect $\diamond\mathcal{P}$ failure detector class, which are referenced in Chapter 5 when comparing to the proposed communication-optimal implementations of $\diamond\mathcal{P}$ failure detectors.

C.1 Communication-Efficient Implementation of $\diamond\mathcal{P}$

In this section we present a basic communication-efficient algorithm implementing a failure detector of the eventually perfect $\diamond\mathcal{P}$ class, which is a non-optimized version of our algorithm of *[Larrea and Lafuente, 2005]* and was presented in *[Larrea et al., 2007b]*.

System Model. We assume the same system model described in Chapter 5; there is a set of processes arranged in a logical ring and that can only fail by crashing. Every pair of processes is connected by two unidirectional and reliable logical links.

Description. In Algorithm C.1 each process sends heartbeats to its successor in the ring, and monitors its predecessor by hearing heartbeats from it. The algorithm works as follows. Every process p starts sending periodically a heartbeat message to its successor in the ring, denoted by the variable $succ_p$ (Task 1). Also, every process p waits for periodical heartbeats from its predecessor in the ring, denoted by the variable $pred_p$. If p does not receive such a heartbeat on a specific time-out interval of $\Delta_p(pred_p)$, then p suspects that $pred_p$ has crashed, includes $pred_p$ in L_p, and sets $pred_p$ to $pred(pred_p)$ (Task 2). If later on p receives a heartbeat message from a process q it is erroneously suspecting, p corrects its local list of suspected processes L_p, increases the time-out interval $\Delta_p(q)$, and changes the perception of its correct predecessor in the ring to q (Task 3). Finally, to avoid scenarios in which several non-intersecting rings could remain forever, every process p periodically sends a heartbeat message to the processes in the ring between itself and $succ_p$ (Task 4).

Besides its local list of suspected processes L_p, every process p has a global list G_p that provides the properties of $\diamond\mathcal{P}$. Processes propagate the global lists around the ring, piggybacked in the heartbeat messages they sent in Task 1 and Task 4. Whenever p

Algorithm C.1: A basic communication-efficient algorithm implementing $\diamond\mathcal{P}$

```
{Every process p executes the following}
1  Procedure main()
2    pred_p ← pred(p)                          {p's estimation of its nearest correct predecessor in the ring}
3    succ_p ← succ(p)                          {p's estimation of its nearest correct successor in the ring}
4    L_p ← ∅ ; G_p ← ∅                         {L_p provides the properties of ◇Q; G_p provides the properties of ◇P }
5    foreach q ∈ Π do Δ_p(q) ← default time-out interval   {Δ_p(q) denotes the duration of p's time-out for q}
6  || Task 1: repeat periodically                                                  {Sending heartbeats}
7    if succ_p ≠ p then send (p-IS-ALIVE, G_p) to succ_p
8  || Task 2: repeat periodically                                                  {Checking time-outs}
9    if ( pred_p ≠ p and p did not receive (pred_p-IS-ALIVE, -)
          during the last Δ_p(pred_p) ticks of p's clock ) then                   {time-out, so p suspects q has crashed}
10     L_p ← L_p ∪ {pred_p}
11     G_p ← G_p ∪ {pred_p}
12     pred_p ← pred(pred_p)
13 || Task 3: when receive (q-IS-ALIVE, G_q) for some q                          {Processing ALIVES}
14   if q ∈ L_p q then                                                          {p was erroneously suspecting q}
15     L_p ← L_p − {succ(pred_p), ··· , q}
16     Δ_p(q) ← Δ_p(q) + 1                                                      {not needed if q = p}
17     pred_p ← q
18   if q = pred_p then
19     G_p ← (G_q − {p}) ∪ L_p
20     succ_p ← p's nearest process following the ring such that ∉ G_p
21 || Task 4: repeat periodically                                               {Set the right sucessor}
22   if succ_p ≠ succ(p) then
23     send (p-IS-ALIVE, G_p) to succ(p), ··· , pred(succ_p)
```

includes a process q in L_p (Task 2), p also includes q in G_p. Finally, each time p receives a heartbeat message from its supposed correct predecessor ($pred_p$) in the ring (Task 3), p builds a new global list of suspected processes by merging the global list G_{pred_p} carried by the heartbeat (removing p if $p \in G_{pred_p}$) with its own local list L_p. Also, p sets its supposed correct successor ($succ_p$) in the ring to its nearest process following the ring not belonging to G_p.

Note that Task 1 and Task 4 could be integrated into a single task in which a process p periodically sends a heartbeat message to all processes in $\{succ(p), \ldots, succ_p\}$. Having two separated tasks, besides providing a clearer presentation, allows the use of independent periods for them, which can be of interest from a performance point of view.

C.2 Basic Implementation of $\diamond\mathcal{P}$ Proposed in the Failure Detectors Seminal Paper

[Chandra and Toueg, 1996] proposed a simple implementation of the $\diamond\mathcal{P}$ failure detector class in partially synchronous systems.

System model. We assume the same system model as in the previous case.

Algorithm C.2 shows the algorithm. It is a time-out based implementation which follows an all-to-all communication pattern. More precisely, every process sends an ALIVE to all the processes. When a process p do not receive an ALIVE message from a process q for an expected time, p will suspect q and hence add it to its suspicions list. If the expected message is finally received, p will remove q from its suspicions list and will increase the waiting time for that process.

Algorithm C.2: A basic implementation of the $\diamond\mathcal{P}$ failure detector class proposed by Chandra and Toueg

```
1  Procedure main()
2      output_p ← ∅
3      foreach q ∈ Π do Δ_p(q) ← default time-out interval      {Δ_p(q) denotes the duration of p's time-out for q}
4  || Task 1: repeat periodically                               {Sending heartbeats}
5      send (p-IS-ALIVE) to all
6  || Task 2: repeat periodically                               {Checking time-outs}
7      foreach q ∈ Π do
8          if ( q ∉ output_p and p did not receive (q-IS-ALIVE)
               during the last Δ_p(q) ticks of p's clock )      then   {p times-out on q so p suspects q has crashed}
9              output_p ← output_p ∪ {q}
10 || Task 3: when receive (q-IS-ALIVE) for some q              {Processing ALIVES}
11     if q ∈ L_p q then                                        {p knows that it prematurely timed-out on q}
12         output_p ← output_p − {q}                            {p repents on q}
13         Δ_p(q) ← Δ_p(q) + 1                                  {p increases its time-out period for q}
```

Bibliography

Marcos Kawazoe Aguilera and Michel Raynal. The 2010 Edsger W. Dijkstra Prize in Distributed Computing. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2010. ISBN 978-3-642-15762-2.

Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In Prasad Jayanti, editor, *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 1999. ISBN 3-540-66531-5.

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

Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In Jennifer L. Welch, editor, *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 108–122. Springer, October 2001. ISBN 3-540-42605-1.

Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Ω with weak reliability and synchrony assumptions. In *PODC*, pages 306–314, July 2003.

Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In Soma Chaudhuri and Shay Kutten, editors, *PODC*, pages 328–337. ACM, 2004. ISBN 1-58113-802-4.

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

Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

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

Sergio Arévalo, Ernesto Jiménez, Mikel Larrea, and Luis Mengual. Communication-efficient and crash-quiescent omega with unknown membership. *Inf. Process. Lett.*, xx(z):xx–yy, To appear in 2011.

Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. ISBN 0471453242.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

Gildas Avoine and Serge Vaudenay. Optimal fair exchange with guardian angels. In Kijoon Chae and Moti Yung, editors, *WISA*, volume 2908 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003. ISBN 3-540-20827-5.

Gildas Avoine, Felix C. Gärtner, Rachid Guerraoui, and Marko Vukolic. Gracefully degrading fair exchange with security modules. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2005. ISBN 3-540-25723-3.

Özalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Software Eng.*, 27(4):308–336, 2001.

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

Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In Özalp Babaoglu and Keith Marzullo, editors, *WDAG*, volume 1151 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 1996. ISBN 3-540-61769-8.

Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30, 1983.

Zinaida Benenson, Milan Fort, Felix C. Freiling, Dogan Kesdogan, and Lucia Draque Penso. TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 34–48. Springer, September 2006a. ISBN 3-540-44601-X.

Zinaida Benenson, Felix C. Freiling, Thorsten Holz, Dogan Kesdogan, and Lucia Draque Penso. Safety, liveness, and information flow: Dependability revisited. In Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, and Erik Maehle, editors, *ARCS Workshops*, volume 81 of *LNI*, pages 56–65. GI, 2006b. ISBN 3-88579-175-7.

Manuel Blum and Shafi Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In *CRYPTO*, pages 289–302, 1984.

Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

certgate GmbH. certgate Smart Card. Internet: `http://www.certgate.com/web_en/products/smartcardmmc.html`, 2008.

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proc. of the 10th ACM symposium on Principles of Distributed Computing (PODC)*, pages 325–340, 1991.

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. A preliminary version appeared in *[Chandra and Toueg, 1991]*.

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

Bernadette Charron-Bost. Comparing the atomic commitment and consensus problems. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 29–34. Springer, 2003. ISBN 3-540-00912-4.

Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 552–565. Springer, 2000. ISBN 3-540-67897-2.

Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

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

Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201703297.

Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In Jennifer Rexford and Emin Gün Sirer, editors, *NSDI*, pages 153–168. USENIX Association, 2009.

Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. The design of a COTSReal-Time distributed security kernel. In Fabrizio Grandoni 0002 and Pascale Thévenod-Fosse, editors, *EDCC*, volume 2485 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2002. ISBN 3-540-00012-7.

Roberto Cortiñas, Felix C. Freiling, Marjan Ghajar-Azadanlou, Alberto Lafuente, Mikel Larrea, Lucia Draque Penso, and Iratxe Soraluze. Secure Failure Detection in TrustedPals. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *SSS*, volume 4838 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. ISBN 978-3-540-76626-1.

Roberto Cortiñas, Iratxe Soraluze, Alberto Lafuente, and Mikel Larrea. Brief announcement: an efficient failure detector for omission environments. In Andréa W. Richa and Rachid Guerraoui, editors, *PODC*, pages 83–84. ACM, 2010. ISBN 978-1-60558-888-9.

George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, 4/e*. Addison-Wesley, 4 edition, June 2005. ISBN-10: 0321263545 | ISBN-13: 9780321263544.

Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999. URL `http://www-cse.ucsd.edu/users/cfetzer/MODEL/`.

George Danezis and Claudia Diaz. A survey of anonymous communication channels. Technical Report MSR-TR-2008-35, Microsoft Research, 2008.

Xavier Défago. Atomic broadcast. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, page 1166. Springer, 1 edition, August 2008. ISBN 978-0-387-30162-4. ISBN-10: 0387307702 ISBN-13: 978-0387307701.

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

Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In Soma Chaudhuri and Shay Kutten, editors, *PODC*, pages 338–346. ACM, 2004. ISBN 1-58113-802-4.

Carole Delporte-Gallet, Hugues Fauconnier, and Felix C. Freiling. Revisiting failure detection and consensus in omission failure environments. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2005. ISBN 3-540-29107-5.

Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling, Lucia Draque Penso, and Andreas Tielmann. From crash-stop to permanent omission: Automatic transformation and weakest failure detectors. In Andrzej Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 165–178. Springer, 2007. ISBN 978-3-540-75141-0.

Carole Delporte-Gallet, Hugues Fauconnier, Andreas Tielmann, Felix C. Freiling, and Mahir Kilic. Message-efficient omission-tolerant consensus with limited synchrony. In *IPDPS*, pages 1–8. IEEE, 2009.

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

Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical report, -, Ithaca, NY, USA, 1996.

Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *PODC*, page 286, 1997. From *[Dolev et al., 1996]*.

Assia Doudou, Benoît Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In Jan Hlavicka, Erik Maehle, and András Pataricza, editors, *EDCC*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999. ISBN 3-540-66483-1.

Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In Johann Blieberger and Alfred Strohmeier, editors, *Ada-Europe*, volume 2361 of *Lecture Notes in Computer Science*, pages 24–50. Springer, 2002. ISBN 3-540-43784-3.

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

Antonio Fernández, Ernesto Jiménez, and Sergio Arévalo. Minimal system conditions to implement unreliable failure detectors. In *PRDC*, pages 63–72. IEEE Computer Society, 2006. ISBN 0-7695-2724-8.

Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

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

Felix C. Freiling and Hagen Völzer. Illustrating the impossibility of crash-tolerant consensus in asynchronous systems. *Operating Systems Review*, 40(2):105–109, 2006.

Felix C. Freiling, Rachid Guerraoui, and Petr Kouznetsov. The failure detector abstraction. *ACM Computing Surveys (CSUR)*, 43:9:1–9:40, February 2011. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1883612.1883616. URL `http://doi.acm.org/10.1145/1883612.1883616`.

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

Oded Goldreich. Secure multi-party computation. Internet: `http://www.wisdom.weizmann.ac.il/~oded/pp.html`, 2002.

Jim Gray. Notes on data base operating systems. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978. ISBN 3-540-08755-9.

Fabíola Greve, Michel Hurfin, Raimundo A. Macêdo, and Michel Raynal. Consensus based on strong failure detectors: A time and message-efficient protocol. In José D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1258–1265. Springer, 2000. ISBN 3-540-67442-X.

Rachid Guerraoui. Revistiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *WDAG*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 1995. ISBN 3-540-60274-7.

Rachid Guerraoui. Indulgent algorithms. In *PODC*, pages 289–297, 2000.

Rachid Guerraoui. Failure detectors. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, page 1166. Springer, 1 edition, August 2008. ISBN 978-0-387-30162-4. ISBN-10: 0387307702 ISBN-13: 978-0387307701.

Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.

Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540288457.

Rachid Guerraoui, Mikel Larrea, and André Schiper. Non-blocking atomic commitment with an unreliable failure detector. In *SRDS*, pages 41–50, 1995.

Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In Sacha Krakowiak and Santosh K. Shrivastava, editors, *Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1999. ISBN 3-540-67196-X.

Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 399–412. Springer, 2006. ISBN 3-540-44624-9.

Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.

Vassos Hadzilacos and Sam Toueg. Distributed systems (2nd edition). chapter Fault-tolerant broadcasts and related problems, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition, 1993. ISBN 0-201-62427-3. Expanded version appeared as a Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca, NY, 1994. ISBN 0-201-62427-3.

Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425 / TR11-84, 1994. URL `citeseer.ist.psu.edu/hadzilacos94modular.html`. Technical Report from his Ph.D. thesis *[Hadzilacos, 1984]* and *[Hadzilacos and Toueg, 1993]*.

Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for byzantine fault detection. In *Second Workshop on Hot Topics in System Dependability (HotDep '06)*, 2006.

Marit Hansen and Andreas Pfitzmann. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management - a consolidated proposal for terminology. Internet: `http://dud.inf.tu-dresden.de/Anon_Terminology.shtml`, February 2008.

Maurice Herlihy. Asynchronous consensus impossibility. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, page 1166. Springer, 1 edition, August 2008. ISBN 978-0-387-30162-4. ISBN-10: 0387307702 ISBN-13: 978-0387307701.

Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6): 858–923, 1999.

Maurice Herlihy and J. D. Tygar. How to make replicated data secure. In Carl Pomerance, editor, *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 379–391. Springer, 1987. ISBN 3-540-18796-0.

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

Martin Hutle. *Failure detection in Sparse Networks*. PhD thesis, Technischen Universitat Wien Fakultat fur Informatik, 2005.

Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 4 1994. ISBN 0133013677.

Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *PODC*, pages 75–84. ACM, 2008. ISBN 978-1-59593-989-0.

Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit at no additional cost. In *PODS*, pages 245–254. ACM Press, 1995. ISBN 0-89791-730-8.

Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

Vladimir Kolesnikov. Gate evaluation secret sharing and one-round two-party computation. In *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

Louis Kruger, Somesh Jha, Eu-Jin Goh, and Dan Boneh. Secure function evaluation with ordered binary decision diagrams. In *ACM CCS*, pages 410–420. ACM Press, 2006.

Alberto Lafuente, Mikel Larrea, Iratxe Soraluze, and Roberto Cortiñas. Brief announcement: Optimal failure detection with low sporadic overhead and communication locality. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *PODC*, page 450. ACM, 2008. ISBN 978-1-59593-989-0.

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

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

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

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

Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical report, 1982.

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

J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992. ISBN 0387822968.

Mikel Larrea and Alberto Lafuente. Brief announcement: Communication-efficient implementation of failure detector classes $\diamond\mathcal{Q}$ and $\diamond\mathcal{P}$. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 495–496. Springer, 2005. ISBN 3-540-29163-6.

Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In Prasad Jayanti, editor, *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 1999. ISBN 3-540-66531-5.

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

Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Eventually consistent failure detectors. *Journal of Parallel and Distributed Computing*, 65(3):361–373, 2005.

Mikel Larrea, Alberto Lafuente, Iratxe Soraluze, Roberto Cortiñas, and Joachim Wieland. On the implementation of communication-optimal failure detectors. In Andrea Bondavalli, Francisco Vilar Brasileiro, and Sergio Rajsbaum, editors, *LADC*, volume 4746 of *Lecture Notes in Computer Science*, pages 25–37. Springer, 2007a. ISBN 978-3-540-75293-6.

Mikel Larrea, Alberto Lafuente, Iratxe Soraluze, Roberto Cortiñas, and Joachim Wieland. Designing efficient algorithms for the eventually perfect failure detector class. *Journal of Software (JSW)*, 2(4):1–11, October 2007b. URL `http://www.academypublisher.com/ojs/index.php/jsw/index`.

Mikel Larrea, Iratxe Soraluze, Roberto Cortiñas, and Alberto Lafuente. An evaluation of communication-optimal $\diamond\mathcal{P}$ algorithms. In *PDP*, pages 274–279. IEEE Computer Society, 2008.

Neal Leavitt. Will proposed standard make mobile phones more secure? *IEEE Computer*, 38(12):20–22, 2005.

Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer-Verlag, 2007.

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.

P. MacKenzie, A. Oprea, and M.K. Reiter. Automatic generation of two-party computations. In *SIGSAC: 10th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2003.

Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — A secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004. URL `http://www.cs.huji.ac.il/~noam/fairplay.pdf`.

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

Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

Achour Mostéfaoui and Michel Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In Prasad Jayanti, editor, *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 1999. ISBN 3-540-66531-5.

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

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

Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *PODC*, pages 281–293, 1987.

Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334–367, 1993. Previous version in *[Neiger and Toueg, 1987]*.

Henning Pagnia, Holger Vogt, and Felix C. Gärtner. Fair exchange. *The Computer Journal*, 46(1), 2003.

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

Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, March 1986. ISSN 0178-2770.

Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/1052796.1052806.

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

Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

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

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

Fred B. Schneider. What good are models and what models are good? pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3. ISBN:0-201-62427-3.

Dale Skeen. Nonblocking commit protocols. In Y. Edmund Lien, editor, *SIGMOD Conference*, pages 133–142. ACM Press, 1981.

Iratxe Soraluze, Roberto Cortiñas, Alberto Lafuente, Mikel Larrea, and Felix C. Freiling. Communication-Efficient Failure Detection and Consensus in Omission Environments. *Information Processing Letters*, 111(6):262–268, February 2011. ISSN 0020-0190. doi: DOI:10.1016/j.ipl.2010.12.008. URL `http://www.sciencedirect.com/science/article/B6V0F-51N22D0-5/2/6a81f701f22b52609ea5bb4cef734d8e`.

Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo, and William H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *SRDS*, pages 71–82. IEEE Computer Society, 2006. ISBN 0-7695-2677-2.

John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25 (6):8–17, 1992.

Paulo Veríssimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

Josef Widder and Ulrich Schmid. The Theta-Model: Achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

Weigang Wu, Jiannong Cao, Jin Yang, and Michel Raynal. A hierarchical consensus protocol for mobile ad hoc networks. In *PDP*, pages 64–72. IEEE Computer Society, 2006. ISBN 0-7695-2513-X.

Andrew C. Yao. Protocols for secure computations. In *Proceedings of the Twenty-Third Annual Symposium on Foundations of Computer Science*, pages 160–164. Springer-Verlag, November 1982.

Glossary

Glossary
Symbols

Π set of processes in the system. 7
Θ see theta. 13
c number of correct processes in the system. 7, 100
f number of faulty processes in the system. 7
n number of processes in the system. 7
Ω the Omega failure detector class. 27, 121
$\diamond\mathcal{P}$ the Eventually Perfect failure detector class. 27
$\diamond\mathcal{Q}$ the Eventually Quasi Perfect failure detector class. 27
$\diamond\mathcal{S}$ the Eventually Strong failure detector class. 27
$\diamond\mathcal{W}$ the Eventually Weak failure detector class. 27
$\mathcal{P}$ the Perfect failure detector class. 27
$\mathcal{Q}$ the Quasi Perfect failure detector class. 27
$\mathcal{S}$ the Strong failure detector class. 27
$\mathcal{W}$ the Weak failure detector class. 27
***k*-set agreement** . 20, 102

A

accuracy (property of failure detectors) capability of avoiding mistakes when suspecting faulty processes. 25
algorithm . 7
asynchronous . 11–13, 18

B

b-link communication abstraction. 86, 101
Byzantine (failure model). 15, 72, 100, 101

C

clock . 10
communication efficiency . i, 31, 32, 34, 86, 100, 120
communication optimality . i, 31, 34, 100, 101, 120
communication-efficient see communication efficiency. 34, 98, 101, 109
communication-optimal see communication optimality. i, 109
completeness (property of failure detectors) capability of suspecting faulty processes. 25
condition See property. 6
consensus processes agreeing on a common value. 17, 66

crash (failure model) failure model that represents process *crash*es, a type of failure which implies that the involved process will not be able to execute any more steps, including local computation and sending or receiving messages. The process is supposed to keep this state forever; however, it there is a possibility for recovering and participating again in the distributed protocol, then the crash-recovery failure model is assumed instead. 14, 23, 24, 31, 33, 86, 100, 101, 121
crash prone see crash. 14
crash-recovery (failure model) processes may crash and later recover. 14, 121
crash-stop see crash. 14
cryptosystem asymmetric key encryption algorithm first proposed by *[Blum and Goldwasser, 1984]*. 84

D

dependability ability to avoid service failures that are more frequent and more severe than is acceptable. 1

E

equivalent . 28
event . 7, 10
eventually synchronous . 12

F

fail-stop (failure type/model). 14
failure detector abstract module located at each process of the system that provides (unreliable) information about (the state of) other processes of the system. 25, 120–122
failure detector history . 25, 60
failure pattern . 24
FLP impossibility result showed by *F*ischer, *L*ynch and *P*aterson. 19, 20

G

general omission (failure model) processes may suffer omissions at sending or at receiving. 14, 52, 55, 72, 84, 86, 87, 98, 100, 101
global clock . 10, 24
GST Global *S*tabilization *T*ime. 12

I

indulgent (indulgence) property related failure detectors achieving uniform solutions. 28

L

liveness (property). 6, 9, 12, 18, 19
logical clock . 10
logical time . 10, 11

O

Omega the Omega failure detector class (see Ω). 27

P

partial synchrony . 12
partially synchronous . 12, 13, 20, 33, 55, 88

permanent permanent omission failure type. 15
process basic computational entity. 7, 122
processor See process. 7
property . 6, 120

R

receive-omission message omission failure type that at receiving. 14
Reliable Broadcast reliably sending a message to a set of processes. 107, 108
run (of an algorithm). 8

S

safety (property). 6, 9, 11, 12, 18, 19
scrambler . 84
Secure Multiparty Computation problem presented in *[Yao, 1982]*. i, 4, 72–74, 100, 101, 122
send-omission message omission failure type that at sending. 14
site See process. 7
SMC see Secure Multiparty Computation. iii, 73, 74
stable storage . 14
state . 5
step . 6, 7, 10
synchronous . 11–13, 18
system model set of properties met by a system. Alternative definition (*[Schneider, 1993]*): collection of attributes and set of rules that govern how these attributes interact. 5

T

theta a partially synchronous model. 13, 120
time-free . 11
timing failure (failure type). 15, 55
trace . 6
transient transient omission failure type. 15
TrustedPals . iii, 4, 72–74, 100, 101

U

uniform see uniformity. 18, 79
uniformity . 6, 122

W

weaker relation between failure detectors in order to compare them. 28
weakest (failure detector to solve a problem) function that provides the *minimal* failure detector that solves a given problem. 28