SYNCHRONIZATION USING FAILURE DETECTORS

THÈSE N^{O} 3262 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Petr KOUZNETSOV

M.Sc. in Mathematics, Saint-Petersburg State University of Information Technology, Mechanics and Optics, Russie
et de nationalité russe

acceptée sur proposition du jury:

Prof. R. Guerraoui, directeur de thèse
Prof. V. Hadzilacos, rapporteur
Prof. E. Ruppert, rapporteur
Prof. M. Shokrollahi, rapporteur

Lausanne, EPFL
2005

Abstract

Many important synchronization problems in distributed computing are impossible to solve (in a fault-tolerant manner) in purely *asynchronous systems*, where message transmission delays and relative processor speeds are unbounded. It is then natural to seek for the minimal *synchrony assumptions* that are sufficient to solve a given synchronization problem. A convenient way to describe synchrony assumptions is using the *failure detector* abstraction.

In this thesis, we determine the weakest failure detectors for several fundamental problems in distributed computing: solving fault-tolerant mutual exclusion, solving non-blocking atomic commit, and boosting the synchronization power of atomic objects. We conclude the thesis by a perspective on the very definition of failure detectors.

Résumé

De nombreux problèmes importants en informatique répartie sont impossibles à résoudre (de manière tolérante aux défaillances) au sein des systèmes purement *asynchrones*, dans lesquels le temps pour transmetre un message et les vitesses relatives des processeurs sont illimitées. Il est naturel de chercher les conditions minimales de synchronisme qui sont suffisantes pour résoudre un problème reparti donné. Une manière commode de décrire ces conditions consiste à utiliser l'abstraction de détecteur de fautes.

Cette thèse détermine les détecteurs de fautes les plus faibles pour plusieurs problèmes fondamentaux de l'informatique répartie: résoudre l'exclusion mutuelle de manière tolérante aux fautes, résoudre la validation atomique de manière non-bloquante, et amplifier la puissance de synchronisation des objets atomiques. Nous concluons la thèse par une perspective sur la définition même de la notion de détecteurs de fautes.

Acknowledgements

During my Ph.D. term, I was very lucky to meet and to learn from a bunch of wonderful people.

First of all, I am very thankful to my supervisor, Prof. Rachid Guerraoui, whose generosity and enthusiasm has been invaluable for my work and life in general.

I would like to express my thanks to the members of my Ph.D. jury, Prof. Vassos Hadzilacos, Prof. Eric Ruppert, and Prof. Amin Shokrollahi, and to the president of the jury Prof. Emre Telatar, for spending their time on my thesis and providing lots of helpful comments.

I am especially grateful to Sidath Handurukande for being a great friend and office-mate, and Arnas Kupsys for his beautiful friendship. Many thanks to Partha Dutta for a lot of inspiring discussions.

I would also like to gratefully mention other fellows in the Distributed Programming Lab, including Sébastien Baehni, Ron Levy, Bastian Pochon, and, especially, our secretary Kristine Verhamme.

I am very grateful to all people from the research community with whom I had an honor to communicate and collaborate. These include Hagit Attiya, Carole Delporte-Gallet, Patrick Th. Eugster, Hugues Fauconnier, Eli Gafni, Felix Gärtner, Vassos Hadzilacos, Nancy Lynch, Luis Rodrigues, Eric Ruppert, and Sam Toueg. I am also grateful to the anonymous conference and journal reviewers for their comments on the material presented in this thesis. I am especially thankful to Prof. Sergey A. Vavilov for giving me a taste for research.

Many thanks should go to the organizers of the Doctoral School program at the School of Computer and Communication Sciences of EPFL, and to the teachers of the School. Very special thanks to Pavel Balabko who introduced me to the Doctoral School program.

Preface

This thesis is devoted to the use of failure detectors in solving synchronization problems and, in particular, to the weakest failure detector question. The main results of this thesis appeared originally in the following papers:

1. R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, pages 461–473, August 2002.

2. R. Guerraoui and P. Kouznetsov. The weakest failure detector for non-blocking atomic commit. Technical Report IC/2003/47, EPFL, May 2003.

3. R. Guerraoui and P. Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, October 2003.

4. R. Guerraoui and P. Kouznetsov. The gap in circumventing the consensus impossibility. In submission. Technical report, EPFL, ID:IC/2004/28, January 2004.

5. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 338–346, July 2004.

6. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Dustributed Computing (JPDC)*, 65(4):492–505, April 2005.

Concurrently, I was involved in a number of research projects, including (i) the design and analysis of gossip-based broadcast algorithms, that resulted in a novel Lightweight Probabilistic Broadcast algorithm (*lpbcast*) and Δ-Reliability, a novel probabilistic measure for reliability of broadcast algorithms, (ii) examining the limitations of composing fault-tolerant distributed services, and (iii) exploring the intersection of distributed computing and algebraic topology. This involvement resulted in the following papers:

1. R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Reducing noise in gossip-based reliable broadcast. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS 2001)*, October 2001.

2. P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, November 2003.

3. P. T. Eugster, R. Guerraoui, and P. Kouznetsov. Delta-Reliable Broadcast: A Probabilistic Measure of Broadcast Reliability. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.

4. P. C. Attie, R. Guerraoui, P. Kouznetsov, N. A. Lynch, and S. Rajsbaum. The impossibility of boosting distributed service resilience. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, June 2005.

5. R. Guerraoui, P. Kouznetsov, and B. Pochon. A note on set agreement with omission failures. In *Proceedings of the 4th Workshop on Geometric and Topological Methods in Concurrency and Distributed Systems Theory (GETCO 2002)*, Toulouse, October 2002.

6. S. Blanc, R. Guerraoui, K. Hess, P. Kouznetsov, P. E. Parent, B. Pochon, and O. Sauvageot. Using the topological characterization of synchronous models. In *Proceedings of the 4th Workshop on Geometric and Topological Methods in Concurrency and Distributed Systems Theory (GETCO 2002)*, Toulouse, October 2002.

7. R. Guerraoui, P. Kouznetsov, and B. Pochon. On the asynchronous computability theorem. In *Proceedings of the 6th Workshop on Geometric and Topological Methods in Concurrency and Distributed Systems Theory (GETCO 2004)*, Amsterdam, October 2004.

Contents

1 Introduction **1**

1.1 Distributed computing and synchronization 1

1.1.1 Consensus 2

1.1.2 Non-blocking atomic commit 2

1.1.3 Weak consensus 2

1.1.4 Fault-tolerant mutual exclusion 2

1.2 Distributed computing model 3

1.2.1 Asynchronous system 3

1.2.2 Impossibility of synchronization 3

1.2.3 Partially synchronous models 3

1.2.4 Asking an oracle 4

1.3 The failure detector abstraction 4

1.3.1 Overview 4

1.3.2 Comparing failure detectors 5

1.3.3 The weakest failure detector question 6

1.4 Main results of this thesis 6

1.4.1 Mutual exclusion 6

1.4.2 Quittable consensus and non-blocking atomic commit . . . 7

1.4.3 Boosting the consensus power with failure detectors 8

1.5 Outline of this thesis 9

2 Model **11**

2.1 Processes 11

2.2 Failures and failure patterns 11

2.3 Failure detectors 12

2.4 Algorithms 13

2.5 Configurations, schedules, and runs 13

2.6 Problems and solvability 14

2.7 Consensus and weak consensus 15

2.8 The merging lemma 15

2.9 Causality and the initial state lemma 16
2.10 Reducibility 16
2.11 A weakest failure detector 17

3 Background 19

3.1 The CHT proof 19
3.1.1 Overview of the reduction algorithm 19
3.1.2 Building a DAG 20
3.1.3 Simulation trees 21
3.1.4 Tags and valences 22
3.1.5 Stabilization 22
3.1.6 Critical index 23
3.1.7 The reduction algorithm 26
3.1.8 Multivalent critical index 27
3.1.9 Weak consensus vs. consensus 28
3.2 Implementing a register 28
3.2.1 Read/write shared memory 29
3.2.2 The sufficiency part 29
3.2.3 The reduction algorithm 29
3.2.4 Solving consensus in all environments 31

4 Mutual Exclusion 33

4.1 The fault-tolerant mutual exclusion problem 33
4.2 The trusting failure detector 34
4.3 The necessary condition for solving FTME 37
4.4 The sufficient condition for solving FTME 40
4.5 On the number of correct processes 44
4.6 Group mutual exclusion 46
4.7 Cost of resilience 50
4.8 Implementing $\mathcal{T}$ 51
4.9 Open questions 52
4.10 Related work 52

5 Quittable Consensus and NBAC 55

5.1 Quittable consensus (QC) 55
5.2 The weakest failure detector to solve QC 56
5.2.1 Specification of failure detector Ψ 57
5.2.2 Using Ψ to solve QC 57
5.2.3 Extracting Ψ from any failure detector that solves QC . . . 59
5.3 The weakest failure detector to solve NBAC 65

5.3.1 Specification of NBAC 65
5.3.2 Using $\mathcal{FS}$ to relate NBAC and QC 66
5.3.3 The weakest failure detector to solve NBAC 67
5.4 Concluding remarks 68
5.5 Related work 69

6 Failure Detectors as Type Boosters 71
6.1 Preliminaries 71
6.1.1 Objects and types 71
6.1.2 Algorithms 72
6.1.3 Configurations, schedules and runs 73
6.1.4 Consensus and consensus power 74
6.1.5 Team consensus 74
6.1.6 Weak consensus 75
6.2 Hierarchy of failure detectors Ω_n 76
6.3 Boosting consensus power to level $n + 1$ 76
6.3.1 Overview of the reduction algorithm 77
6.3.2 DAGs and simulation trees 78
6.3.3 Decision gadgets 79
6.3.4 Critical index 82
6.3.5 Complete decision gadgets 83
6.3.6 Confused processes 83
6.3.7 Deciding sets 85
6.3.8 Reduction algorithm 87
6.4 Boosting consensus power to any level $k > n$ 90
6.5 Boosting object resilience 91
6.5.1 Boosting resilience of atomic objects is impossible 91
6.5.2 Boosting resilience of atomic objects with failure detectors . 93
6.6 Open questions 93
6.7 Related work 94

7 Concluding Remarks 95
7.1 Failure detectors as distributed services 95
7.1.1 I/O automata 95
7.1.2 Distributed services 97
7.1.3 Failure detectors 98
7.1.4 Reliable channels 103
7.1.5 Atomic objects 103
7.1.6 Distributed system model 104
7.1.7 A weakest failure detector 106

7.1.8 Applications 106
7.2 Future directions 106
7.3 Summary 107

Bibliography 114

List of Figures 116

Curriculum Vitae 117

Chapter 1

Introduction

A *distributed system* is a collection of individual computing units that can communicate with each other. This very general definition encompasses a wide spectrum of modern computing systems, including multiprocessors, local-area clusters of workstations, and the Internet.

Distributed computing pursues a variety of benefits, including effective sharing of resources, high availability, and *fault-tolerance*, which is the principal subject of this thesis. However, implementing distributed systems that enjoy these benefits is notoriously difficult.

For example, many fundamental synchronization problems in distributed computing, such as consensus, non-blocking atomic commit and mutual exclusion, are impossible to solve (in a fault-tolerant manner) in purely asynchronous systems [28, 24], where message transmission delays and relative processor speeds are unbounded. It is thus appealing to seek for the minimal *synchrony assumptions* that are sufficient to solve a given synchronization problem. The *failure detector* abstraction [15, 14] provides a convenient way to describe synchrony assumptions. In this chapter, we informally pose the *weakest failure detector* question and overview the main results of this thesis.

1.1 Distributed computing and synchronization

Consider a computing unit that provides a certain service to a set of *clients* by (a) receiving requests from the clients, (b) processing the requests, and (c) sending back the corresponding responses. A failure of the unit makes the whole service unavailable. A standard solution to avoid the issue of a single point of failure would be to *replicate* the service, i.e., to devise a *distributed algorithm* that, running on a collection of computers, would create an illusion of a centralized service to the clients [50]. The components of the replicated system must be *synchronized* in a consistent way that would ensure progress and safety even when some of the components fail by *crashing* (prematurely stopping taking steps of computation). A consistency criterion of such a synchronization (a *synchronization problem*) depends on the type of service that the replicated system is supposed to provide. For example, consider the following classical synchronization problems that arise in fault-tolerant distributed computing: implementing state machines, processing transactions, and managing indivisible resources.

1.1.1 Consensus

The synchronization requirements of the state machine replication problem [50, 66] are captured by *consensus* [28], probably the most studied problem in distributed computing. Informally, in consensus, each thread of computation, which we call a *process*, initially proposes a value, and eventually processes must reach a common decision on one of the proposed values. Using reliable message-passing channels and multiple instances of consensus, it is straightforward to implement a replicated state machine, in which client requests and the corresponding responses are totally ordered [15, 52].

1.1.2 Non-blocking atomic commit

The *non-blocking atomic commit* problem (NBAC) arises in distributed transaction processing [31, 68]. Informally, the set of processes that participate in a transaction must agree on whether to commit or abort that transaction. Initially, each process votes Yes ("I am willing to commit") or No ("we must abort"), and eventually processes must reach a common decision, Commit or Abort. The decision to Commit can be reached only if all processes voted Yes. Furthermore, if all processes voted Yes and no failure occurs, then the decision *must* be Commit.

1.1.3 Weak consensus

Consensus and NBAC are two instances of *agreement* problems. In these problems, processes should eventually agree on a common value. The weakest nontrivial agreement problem is probably the *weak consensus* problem [28], in which processes are required to decide on a common value in $\{0, 1\}$ so that there is an execution in which 0 is decided, and there is an execution in which 1 is decided. We can immediately see that weak consensus is *weaker* than both consensus and NBAC in the sense that any solution to consensus or NBAC can be easily transformed into a solution to weak consensus.

1.1.4 Fault-tolerant mutual exclusion

The *mutual exclusion* problem [23, 51] involves managing a single, indivisible resource that can be accessed by at most one process at a time (*mutual exclusion* property). The process accessing the resource is said to be in its *critical section* (CS). The *fault-tolerant mutual exclusion* problem (FTME) requires that if a non-faulty process wants to enter its CS, then there eventually will be *some* non-faulty process in its CS (*progress* property), even if some process crashes (stops taking steps) while in its CS.

1.2 Distributed computing model

1.2.1 Asynchronous system

Ideally, a distributed algorithm should be correct even if the system is *asynchronous*. In an asynchronous distributed system, messages can take arbitrarily long to be received and processes can run in arbitrarily varying speeds, i.e, there are no bounds on communication (message transmission delays) and processing (relative process speeds). A distributed algorithm that can work even in asynchronous systems would be very appealing in practice, since its correctness would not depend on such bounds.

1.2.2 Impossibility of synchronization

It is well-known that weak consensus (and thus consensus and NBAC) cannot be solved deterministically in asynchronous systems in which at least one process can fail by crashing (Fischer, Lynch and Paterson [28]). Essentially, the proof of [28] is based on the fact that asynchronous systems provide no hints for ever distinguishing a very slow ("sleeping") process from a faulty one.

Similarly, FTME cannot be solved without any information about failures: if a process crashes in its critical section, then no progress condition can be satisfied.

The impossibility result of [28] establishes that asynchronous distributed computability is essentially different from classical Turing computability [69]: the consensus problem, trivially solvable on a single processor, is impossible to solve in an asynchronous system of two or more processors of which one can fail by crashing.

1.2.3 Partially synchronous models

Given that many important synchronization problems, such as consensus, NBAC and FTME, are not solvable in purely asynchronous systems, it is natural to ask what happens if we strengthen the synchrony assumptions. For example, we might want to assume that communication is synchronous, i.e., there exists a known bound on message transmission delays, or, alternatively, that processing is synchronous, i.e., there exists a known bound on relative process speeds. Still, Dolev, Dwork and Stockmeyer [24] showed that restricting only one of these parameters (communication or processing) does not help to solve consensus tolerating just one faulty process.[1]

However, if we restrict *both* parameters, then fault-tolerant consensus becomes solvable. In a very general manner, Dwork, Lynch and Stockmeyer [25] introduced a number of models with different kinds of *partial* synchrony and determined tight bounds on the number of faulty processes that can be tolerated by a consensus algorithm in each of these models.

In particular, they introduced the *partially synchronous* model in every execution of which there are (unknown a priori) bounds on communication and

[1] Concurrently, Loui and Abu-Amara [56] showed that fault-tolerant consensus is impossible to solve in the read/write shared memory model, which is computationally equivalent to the communication synchronous model of [24].

processing.[2] In this model, consensus can be solved tolerating at most a minority of faulty processes. If we strengthen the partially synchronous model by assuming that communication is synchronous (i.e., the message transmission delays are bounded, and the bound is known a priori), and processing is still partially synchronous, then consensus can be solved tolerating any number of faulty processes.

1.2.4 Asking an oracle

Thus, we can circumvent the impossibility of consensus by introducing a certain *amount of synchrony* into the system, or, in other words, by providing a mechanism to reason about process failures.

For example, assume that, in the partially synchronous model, every process p runs the following failure detection algorithm [25, 15]. Periodically, p sends "p-is-alive" to all and then waits for "alive" messages of other processes until a *timeout* expires. If p timeouts on some process q, then p adds q to the list of *suspected* processes. If p later receives a "q-is-alive" message, i.e., p made a mistake by prematurely timing out on q, then p removes q from the list of suspects and increments the timeout, in order to prevent such a mistake in the future. The properties of the partially synchronous model ensure that there is a time after which every non-faulty process will forever suspect every faulty process and stop suspecting any non-faulty process.

The information about failures provided by this algorithm can be abstracted out as follows. Let us consider an asynchronous system in which processes have access to a distributed *oracle*. At each process and at each time, this oracle outputs a list of suspected processes. The oracle guarantees that there is a time after which (a) every faulty process is permanently suspected and (b) no non-faulty process is ever suspected. (In fact, the oracle is an example of a *failure detector*, as we discuss in Section 1.3.)

It turns out that this seemingly weak oracle can dramatically improve the computational power of a distributed system: the synchrony assumptions provided by the oracle are *sufficient* to solve consensus tolerating a minority of faulty processes [25]. If, in addition, communication is synchronous, it is possible to solve consensus tolerating any number of failures [25, 53]. But are these synchrony assumptions also *necessary*? Can we solve fault-tolerant consensus with even less amount of synchrony?

1.3 The failure detector abstraction

1.3.1 Overview

Now we approach the central question of this thesis: what is the *minimal amount of synchrony* that is sufficient to make a given synchronization problem solvable.

To answer the question we must define first what we mean by the "amount of synchrony". Since the impossibility of fundamental synchronization problems inherently comes from the impossibility of reasoning about failures in asynchronous

[2] [25] also considered a variant of the partially synchronous model in which bounds on communication and processing are known a priori, but hold only after some unknown time.

systems [28], it is convenient to describe synchrony in terms of *failure detectors* introduced by Chandra and Toueg [15].

Informally, a failure detector is a distributed oracle that supplies the processes with (possibly incomplete and inaccurate) hints about failures. The output by failure detectors is not restricted in any way, except that it is defined exclusively by failures (e.g., a failure detector cannot leak information about the local variables of other processes). For example, a failure detector can produce at any time a list of identifiers (ids) of processes currently suspected to have crashed; or the id of the current leader process; or a boolean formula, such as "neither p nor q is crashed".

A failure detector can be *unreliable*: it can make an a priori unbounded number of mistakes, e.g., by suspecting non-faulty processes to have crashed or electing a faulty process as a leader.

A number of failure detectors have been defined in the literature. The results of this thesis make use of the following ones:

- The *perfect* failure detector $\mathcal{P}$ outputs a set of *suspected* processes at each process. There is a time after which every crashed process is permanently suspected by every non-faulty process, and no process is ever suspected before it crashes [15].

- The *eventually perfect* failure detector $\diamond\mathcal{P}$ also outputs a set of suspected processes at each process. But the guarantees provided by $\diamond\mathcal{P}$ are weaker than those of $\mathcal{P}$. There is a time after which $\diamond\mathcal{P}$ outputs the set of all faulty processes at every non-faulty process [15].

- The *leader failure detector* Ω outputs the id of a process at each process. There is a time after which it outputs the id of the same non-faulty process at all non-faulty processes [14].

- The *quorum* failure detector Σ outputs a set of processes at each process. Any two sets (output at any times and at any processes) intersect, and eventually every set output at every correct process consists of only non-faulty processes [20].

- The *failure signal* failure detector $\mathcal{FS}$ outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ outputs **green** at every process; after a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every non-faulty process [16, 32].

1.3.2 Comparing failure detectors

Sometimes a distributed algorithm meets its specification only under certain assumptions about when and where failures might occur. These assumptions are represented in the form of *environments* [14]. Examples of environments are: a majority of processes are non-faulty; process p never fails before process q; no process fails after it takes at least one step, etc.

Failure detectors can be partially ordered according to their synchronization power. Informally, we say that a failure detector $\mathcal{D}$ is *weaker* than a failure detector $\mathcal{D}'$ in an environment $\mathcal{E}$ if there exists an algorithm that transforms $\mathcal{D}'$ into $\mathcal{D}$ in $\mathcal{E}$ (the formal definition comes later in Section 2.10). If $\mathcal{D}$ is not weaker than $\mathcal{D}'$ in $\mathcal{E}$ and $\mathcal{D}'$ is not weaker than $\mathcal{D}$ in $\mathcal{E}$, then we say that $\mathcal{D}$ and $\mathcal{D}'$ are *incomparable* in $\mathcal{E}$.

For example, in most environments, Ω is weaker than $\diamond\mathcal{P}$; $\diamond\mathcal{P}$, Σ and $\mathcal{FS}$ are pairwise incomparable; and each of Ω, $\diamond\mathcal{P}$, Σ and $\mathcal{FS}$ is weaker than $\mathcal{P}$.

Note that if $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$, then all problems solvable with $\mathcal{D}$ in $\mathcal{E}$ are also solvable with $\mathcal{D}'$ in $\mathcal{E}$.

1.3.3 The weakest failure detector question

We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ if (a) there is an algorithm that uses $\mathcal{D}$ to solve $\mathcal{M}$ in $\mathcal{E}$, and (b) $\mathcal{D}$ is weaker than any failure detector $\mathcal{D}'$ that can be used to solve $\mathcal{M}$ in $\mathcal{E}$.

Now the central question of this work can be formulated as follows:

> *What is the weakest failure detector to solve a given synchronization problem in a given environment?*

This question has inspired a number of interesting results. Especially related to the results of this thesis are the following ones:

- Ω has been shown to be the weakest to solve consensus if at most a minority of processes can fail [14].
- Σ has been shown to be the weakest to implement read/write shared memory in all environments, i.e., regardless of the number and timing of failures [20].
- (Ω, Σ), the composition of Ω and Σ, has been shown to be the weakest to solve consensus in all environments [20].
- $\mathcal{FS}$ has been shown to be the weakest to solve NBAC with at most one faulty process [16, 32].

1.4 Main results of this thesis

In this thesis, we determine the weakest failure detectors for several fundamental problems in distributed computing. The problems considered are: solving fault-tolerant mutual exclusion, solving quittable consensus, solving non-blocking atomic commit, and boosting the synchronization power of object types.

1.4.1 Mutual exclusion

The mutual exclusion problem [23, 51] involves managing access to a single, indivisible resource that can be accessed by at most one process at a time. In the *fault-tolerant mutual exclusion* problem (FTME), we require that if a non-faulty process wants to access the resource, then access is eventually granted to *some* non-faulty process, even if some process crashes while accessing the resource.

(a) We determine the weakest failure detector to solve FTME with a majority of non-faulty processes. The *trusting* failure detector $\mathcal{T}$ outputs a set of suspected processes at each process and ensures that: (1) there is a time after which $\mathcal{T}$ *trusts* (does not suspect) every non-faulty process, (2) there is a time after which $\mathcal{T}$ permanently suspects any crashed process, and (3) at all times, if $\mathcal{T}$ suspects a process after having trusted it, then the process is crashed. Failure detector $\mathcal{T}$ might suspect temporarily a non-faulty process. Intuitively, $\mathcal{T}$ can thus make mistakes and algorithms using $\mathcal{T}$ are, from a practical point of view, more resilient than those using $\mathcal{P}$.

(b) We show also that a majority of non-faulty processes is necessary for any FTME algorithm using $\mathcal{T}$.

(c) We present a failure detector $(\mathcal{T}, \Sigma)$ which is strictly weaker than $\mathcal{P}$ and which is sufficient (but possibly not necessary) to solve the problem regardless of the number and timing of failures.

(d) We consider *group* mutual exclusion [47, 38], a recent generalization of mutual exclusion, and we show that $\mathcal{T}$ is the weakest to solve *fault-tolerant group mutual exclusion* (FTGME) with a majority of non-faulty processes. Analogously, we show that $(\mathcal{T}, \Sigma)$ is sufficient to solve FTGME regardless of the number and timing of failures.

1.4.2 Quittable consensus and non-blocking atomic commit

Non-blocking atomic commit (NBAC) is a well-known problem that arises in distributed transaction processing [31, 68]. To determine the weakest failure detector to solve NBAC in all environments, we proceed through the following steps:

(a) We consider a natural variation of consensus, called *quittable consensus* (QC) introduced by Hadzilacos and Toueg [40]. Informally, QC is like consensus except that, in case a failure occurs, processes have the option (but not the obligation) to agree on a special value $\mathcal{Q}$ (for "quit"). We determine the weakest failure detector to solve QC in all environments. This failure detector, denoted Ψ, behaves as follows: For an initial period of time, the output of Ψ at each process is $\perp$. Eventually, however, Ψ either behaves like the failure detector (Ω, Σ) at all processes or, if a failure occurs, it may *report a failure* by outputting **red** at all processes. The switch from $\perp$ to behaving like (Ω, Σ) or reporting a failure need not occur simultaneously at all processes, but the same choice is made at all processes.

(b) We establish that NBAC is equivalent to QC modulo the failure detector $\mathcal{FS}$: (i) given $\mathcal{FS}$, any QC algorithm can be transformed into an algorithm for NBAC, and (ii) any algorithm for NBAC can be transformed into an algorithm for QC, and can also be used to implement $\mathcal{FS}$.

(c) We use (a) and (b) to show that $(\Psi, \mathcal{FS})$, the composition of Ψ and $\mathcal{FS}$, is the weakest failure detector to solve NBAC in all environments.

1.4.3 Boosting the consensus power with failure detectors

Fault-tolerant consensus is impossible in asynchronous systems that provide just a simple form of communication (reliable channels or read/write shared memory) [28, 24, 56]. One way to circumvent this impossibility, besides using failure detectors, is to augment the system model with more powerful communication primitives, typically defined through shared object types with sequential specifications [41, 56]. It is convenient to define the power of an object type T, denoted $cons(T)$, as the maximum number n of processes that can solve consensus using only objects of type T and registers. For instance, the power of the register type is simply 1, the power of the test&set type is 2, whereas the compare&swap type has power ∞ (consensus can be solved among any number of processes using compare&swap objects) [41].

In a way, augmenting the system with powerful shared object types is orthogonal to strengthening the synchrony assumptions by using failure detectors. Failure detectors provide information about failures, but cannot be used to communicate information between processes. On the other hand, objects with sequential specifications can be used for inter-process communication, but they do not provide any information about failures.

Neiger proposed in [61] a way to effectively combine these two trends by introducing a hierarchy of failure detectors Ω_n, $n \in \mathbb{N}$, such that:

(i) For all $n, k \in \mathbb{N}$, Ω_n is sufficient to solve consensus among k processes using *any* set of types T such that $cons(T) = n$ and registers.

(ii) For all $n \in \mathbb{N}$, Ω_{n+1} is strictly weaker than Ω_n.

Informally, Ω_n outputs, at each process, a set of processes so that all non-faulty processes eventually detect the same set of at most n processes that includes at least one non-faulty process. Note that Ω_1 is equivalent to Ω.

It was conjectured in [61] that Ω_n is actually the weakest failure detector to boost the power of T to higher levels of the consensus hierarchy. As pointed out in [61], the proof of this conjecture appears to be challenging and was left open.

In this thesis,

(a) We show that, that for all $n, k \in \mathbb{N}$, and all *m-ported one-shot deterministic* types T such that $m \leq n + 1$ and $cons(T) \leq n$, Ω_n is *necessary* to solve consensus among k processes using registers and objects of type T. Although one-shot deterministic types restrict every process to invoke at most one deterministic operation on each object, they include many popular types such as consensus and test&set, and they exhibit complex behavior in the context of the type booster question [42].

As a side result, we formally prove that Ω is necessary to solve consensus using only registers. The result was first stated in [53] but, to our knowledge, its proof has never appeared in the literature.

(b) We consider *t-resilient* implementations of object types (we will simply call these t-resilient objects): a process is only guaranteed to complete its operation on a t-resilient object, as long as no more than t processes crash, where

t is a specified parameter. If more than t processes crash, no operation on a t-resilient object is obliged to return.

We consider a system in which, for some $t, k \in \mathbb{N}$, $t \geq 2$, k processes communicate through (wait-free) registers and t-resilient objects of any types (not necessarily one-shot and deterministic types with a known bound on the number of ports). As a corollary to the results of (a), we show that Ω_{t+1} is the weakest failure detector to solve consensus in this system.

1.5 Outline of this thesis

- In Chapter 2, we present the model of computation, and define formally the notion of a weakest failure detector.
- In Chapter 3, we recall the techniques used to obtain fundamental "weakest failure detector" results.
- In Chapter 4, we determine the weakest failure detector to solve fault-tolerant mutual exclusion with a majority of non-faulty processes.
- In Chapter 5, we determine the weakest failure detector to solve quittable consensus and NBAC in all environments.
- In Chapter 6, we determine the weakest failure detector to boost the consensus power of one-shot deterministic objects and to boost the consensus resilience level of any objects.
- Chapter 7 discusses alternative approaches to defining failure detectors and concludes the thesis by speculating about some future research.

Chapter 2

Model

In this chapter, we describe the asynchronous message-passing model equipped with a failure detector [14], considered in Chapters 4 and 5.

In Chapter 6, we revisit the model and provide additional details that are necessary to describe a model in which processes communicate through atomic objects of a given consensus power. A generic model in which processes communicate through *distributed services*, including reliable channels, atomic objects and failure detectors, is discussed in Chapter 7.

2.1 Processes

The system consists of a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ $(n > 1)$. Every pair of processes is connected by a reliable channel. Processes communicate by message passing. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is a fictional device: the processes have no direct access to it. (The information about global time can however come *indirectly* from failure detectors.) We take the range $\mathbb{T}$ of the clock's ticks to be the set of natural numbers and 0 ($\mathbb{T} = \{0\} \cup \mathbb{N}$).

2.2 Failures and failure patterns

Processes are subject to *crash* failures. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops forever executing any action.

A *failure pattern* F is a function from the global time range $\mathbb{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. Once a process crashes, it does not recover, i.e., $\forall t : F(t) \subseteq F(t+1)$. Let $\mathbb{F}$ denote the set of all failure patterns.

Let $F \in \mathbb{F}$. We define $\mathit{correct}(F) = \Pi - \cup_{t \in \mathbb{T}} F(t)$, the set of *correct* processes in F. Processes in $\Pi - \mathit{correct}(F)$ are called *faulty* in F. A process $p \notin F(t)$ is said to be *alive* at time t. A process $p \in F(t)$ is said to be *crashed* at time t. A failure pattern F such that $\mathit{correct}(F) = \Pi$ is called *failure-free.*

An *environment* $\mathcal{E}$ is a set of failure patterns. $\mathcal{E}_f$ denotes the set of all failure patterns in which up to f processes can crash: $\mathcal{E}_f = \{F \in \mathbb{F} : |\mathit{correct}(F)| \geq$

$n-f\}$. Unless otherwise stated, we consider environments $\mathcal{E}$ in which at least one process might crash and at least one process is correct: $(\forall F \in \mathcal{E} : \mathit{correct}(F) \geq 1) \wedge (\exists F \in \mathcal{E} : \mathit{faulty}(F) \geq 1)$

2.3 Failure detectors

A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. Informally, $H(p,t)$ represents the value output by the failure detector at process p at time t. A *failure detector $\mathcal{D}$* with range $\mathcal{R}_{\mathcal{D}}$ is a function that maps each failure pattern to a *set* of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$. $\mathcal{D}(F)$ is thus the set of failure detector histories permitted by $\mathcal{D}$ for failure pattern F. Note that we do not make any assumption a priori on the range of a failure detector. When any process p performs a step of computation, it can *query* its *failure detector module* of $\mathcal{D}$, denoted $\mathcal{D}_p$, and obtain a value $d \in \mathcal{R}_{\mathcal{D}}$ that encodes some information about failures.

If $\mathcal{D}$ and $\mathcal{D}'$ are failure detectors, $(\mathcal{D}, \mathcal{D}')$ denotes the failure detector that outputs a vector with two components, the first being the output of $\mathcal{D}$ and the second being the output of $\mathcal{D}'$. Formally, $\mathcal{R}_{(\mathcal{D},\mathcal{D}')} = \mathcal{R}_{\mathcal{D}} \times \mathcal{R}_{\mathcal{D}'}$, and for each F, $\tilde{H} \in (\mathcal{D}, \mathcal{D}')(F) \Leftrightarrow \tilde{H} = (H, H')$, $H \in \mathcal{D}(F)$, $H' \in \mathcal{D}'(F)$.

Now we define formally some failure detectors involved in this thesis:

- The *perfect* failure detector $\mathcal{P}$ [15] outputs a set of *suspected* processes at each process. $\mathcal{P}$ ensures *strong completeness*: every crashed process is eventually suspected by every correct process, and *strong accuracy*: no process is suspected before it crashes.

  Formally, for each failure pattern F, and each history $H \in \mathcal{P}(F) \quad \Leftrightarrow$

  $$\Big(\exists t \in \mathbb{T}\ \forall p \in \mathit{faulty}(F)\ \forall q \in \mathit{correct}(F)\ \forall t' \geq t:\ p \in H(q,t')\Big) \wedge$$
  $$\Big(\forall t \in \mathbb{T}\ \forall p, q \in \Pi - F(t):\ p \notin H(q,t)\Big)$$

- The *eventually perfect* failure detector $\Diamond\mathcal{P}$ [15] also outputs a set of suspected processes at each process. But the guarantees provided by $\Diamond\mathcal{P}$ are weaker than those of $\mathcal{P}$. There is a time after which $\Diamond\mathcal{P}$ outputs the set of all faulty processes at every non-faulty process. More precisely, $\Diamond\mathcal{P}$ satisfies strong completeness and *eventual strong accuracy*: there is a time after which no correct process is ever suspected.

  Formally, for each failure pattern F, and each history $H \in \Diamond\mathcal{P}(F) \quad \Leftrightarrow$

  $$\exists t \in \mathbb{T}\ \forall p \in \mathit{correct}(F)\ \forall t' \geq t:\ H(p,t') = \mathit{faulty}(F)$$

- The *leader failure detector* Ω [14] outputs the id of a process at each process. There is a time after which it outputs the id of the same non-faulty process at all non-faulty processes.

  Formally, for each failure pattern F, and each history $H \in \Omega(F) \quad \Leftrightarrow$

  $$\exists t \in \mathbb{T}\ \exists q \in \mathit{correct}(F)\ \forall p \in \mathit{correct}(F)\ \forall t' \geq t:\ H(p,t') = q$$

- The *quorum failure detector* Σ [20] outputs a set of processes at each process. Any two sets (output at any times and at any processes) intersect, and eventually every set consists of only non-faulty processes.

  Formally, for each failure pattern F, and each history $H \in \Sigma(F) \quad \Leftrightarrow$

  $$\big(\forall p, p' \in \Pi \; \forall t, t' \in \mathbb{T} \; H(p,t) \cap H(p',t') \neq \emptyset\big) \wedge$$
  $$\big(\forall p \in \mathit{correct}(F) \; \exists t \in \mathbb{T} \; \forall t' \geq t \; H(p,t') \subseteq \mathit{correct}(F)\big).$$

- The *failure signal* failure detector $\mathcal{FS}$ [16, 32] outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ outputs **green** at every process; after a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every non-faulty process.

  Formally, for each failure pattern F, and each history $H \in \mathcal{FS}(F) \quad \Leftrightarrow$

  $$\big(\forall p \in \Pi \; \forall t \in \mathbb{T} \; \big(H(p,t) = \textbf{red} \rightarrow F(t) \neq \emptyset\big) \wedge$$
  $$\big(\mathit{faulty}(F) \neq \emptyset \rightarrow \forall p \in \mathit{correct}(F) \; \exists t \in \mathbb{T} \; \forall t' \geq t \; H(p,t') = \textbf{red}\big).$$

2.4 Algorithms

The asynchronous communication channels are modeled as a message buffer which contains messages not yet received by their destinations. An *algorithm* $\mathcal{A}$ is a collection of n (possibly infinite state) deterministic automata, one for each process. $\mathcal{A}(p)$ denotes the automaton on which process p is running algorithm $\mathcal{A}$. Computation proceeds in *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p performs atomically the following three actions:

(i) p receives a single message addressed to p from the message buffer, or a null message, denoted λ (*receive* phase);

(ii) p queries and receives a value from its failure detector module (*query* phase);

(iii) p changes its state and sends a message to a single process, according to the automaton $\mathcal{A}(p)$ (*send* phase).

Note that the received message is chosen *non-deterministically* from the messages in the message buffer destined to p, or the null message λ.

2.5 Configurations, schedules, and runs

A *configuration* defines the current state of each process in the system and the set of messages currently in the message buffer. Initially, the message buffer is empty. A step (p, m, d) of an algorithm $\mathcal{A}$ is uniquely determined by the identity of the process p that takes the step, the message m received by p during the step (m might be the null message λ), and the failure detector value d seen by p during the step. We assume that messages are uniquely identified.

We say that a step $e = (p, m, d)$ *is applicable* to a configuration C if and only if $m = \lambda$ or m is in the message buffer of C. For a step e applicable to C, $e(C)$ denotes the unique configuration that results from applying e to C.

A *schedule* S of algorithm $\mathcal{A}$ is a (finite or infinite) sequence of steps of $\mathcal{A}$. $S_\perp$ denotes the empty schedule. We say that a schedule S *is applicable to a configuration* C if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule S applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C.

A *partial run of algorithm* $\mathcal{A}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_\mathcal{D}, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H_\mathcal{D} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is a *finite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is a *finite* list of increasing time values (indicating when each step of S occurred) such that $|S| = |T|$, S is applicable to I, and for all $1 \leq k \leq |S|$, if $S[k] = (p, m, d)$ then:

(1) p has not crashed by time $T[k]$, i.e., $p \notin F(T[k])$, and

(2) d is the value of the failure detector module of p at time $T[k]$, i.e., $d = H_\mathcal{D}(p, T[k])$.

A *run of algorithm* $\mathcal{A}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_\mathcal{D}, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H_\mathcal{D} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is an *infinite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is an *infinite* list of increasing time values indicating when each step of S occurred. In addition to satisfying properties (1) and (2) of a partial run, R should guarantee that

(3) every correct (in F) process takes an infinite number of steps in S and eventually receives every message sent to it.

Property (3) ensures that every correct process must appear in R infinitely often and receive every message sent to it (this conveys the reliability of the communication channels).[1]

Remark. The algorithms presented in Chapters 4 and 5 work (or can be easily transformed to work) also in a weaker model where steps of algorithms have finer granularity (receive phase, query phase, and send phase are not encapsulated in the same atomic step), and channels guarantee only that every correct process eventually receives every message sent to it by any *correct* process. The lower bounds presented in Chapters 4 and 5 hold also in a stronger model in which every process can atomically send its messages to all.

2.6 Problems and solvability

A *problem* is a predicate on a set of runs (usually defined by a set of properties that these runs should satisfy). An algorithm $\mathcal{A}$ *solves a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ if the set of all runs of $\mathcal{A}$ in $\mathcal{E}$ satisfies $\mathcal{M}$. We say

[1] The alternative "I/O automata-based" model, described in Section 7.1 assumes that executions are *fair*: each task gets infinitely many turns to take steps.

that a failure detector $\mathcal{D}$ *solves problem* $\mathcal{M}$ in $\mathcal{E}$ if there is an algorithm $\mathcal{A}$ which solves $\mathcal{M}$ in $\mathcal{E}$ using $\mathcal{D}$.

2.7 Consensus and weak consensus

In the *consensus* problem, each process p starts with an initial value $v \in \{0,1\}$ (we say that p *proposes* v), and terminates with a value $v' \in \{0,1\}$ (we say that p *decides* v'). It is required that:

Termination: Every correct process eventually decides.

Agreement: No two processes (whether correct or faulty) decide different values.

Validity: If a process decides a value v, then v was proposed by some process.

The proof that Ω is necessary to solve consensus presented in [14] holds even for a weaker variant of the problem, called *non-uniform consensus* that requires Termination, Validity and the following form of agreement:

Non-Uniform Agreement: No two *correct* processes decide different values.

The Agreement property of consensus is sometimes referred to as *Uniform Agreement.*

A similar *weak consensus* problem requires Agreement and Termination of consensus, but Validity is substituted with the following weaker property:

Non-Triviality: Every algorithm that solves weak consensus must have a run in which 0 is decided by some process, and a run in which 1 is decided by some process.

Similarly, *non-uniform weak consensus* can be obtained from weak consensus by replacing Agreement with the Non-Uniform Agreement.

By definition, every algorithm that solves (non-uniform) consensus (in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$) trivially solves (non-uniform) weak consensus in $\mathcal{E}$ using $\mathcal{D}$. The converse however does not hold (see Section 3.1.9 for more discussion on this). Since non-uniform weak consensus cannot be solved in asynchronous systems tolerating just one failure [28], neither can any stronger variant of consensus.

2.8 The merging lemma

The following results follow directly from the definition of runs:

Lemma 2.1 *Let $\mathcal{A}$ be any algorithm, and $R_1 = \langle F, H, I, S \cdot S_1, T \cdot T_1 \rangle$ and $R_2 = \langle F, H, I, S \cdot S_2, T \cdot T_2 \rangle$ be partial runs of $\mathcal{A}$, such that the sets of processes that take steps in S_1 and in S_2 are disjoint and T_1 and T_2 contain distinct times. Let $\hat{T}$ be the merging of T_1 and T_2 (in increasing order) and $\hat{S}$ be the corresponding* *merging of S_1 and S_2 (i.e., the steps of $\hat{S}$ are the steps of S_1 and S_2 in the order indicated by $\hat{T}$). Then $\hat{R} = \langle F, H, I, S \cdot \hat{S}, T \cdot \hat{T}\rangle$ is also a partial run of $\mathcal{A}$.*[2]

Corollary 2.2 *Let $\mathcal{A}$ be any weak consensus algorithm, and $R_1 = \langle F, H, I, S \cdot S_1, T \cdot T_1\rangle$ and $R_2 = \langle F, H, I, S \cdot S_2, T \cdot T_2\rangle$ be partial runs of $\mathcal{A}$, such that the sets of processes that take steps in S_1 and in S_2 are disjoint and T_1 and T_2 contain distinct times. If some process decides x_1 in R_1 and some process decides x_2 in R_2 then $x_1 = x_2$.*

2.9 Causality and the initial state lemma

Let $R = \langle F, H, I, S, T\rangle$ be a (partial or regular) run of an algorithm $\mathcal{A}$. Following [50], we define a *causality* relation $\rightarrow_R$ for steps of R as follows. For any $1 \leq k \leq |S|$, let $M[k]$ denote the message buffer of configuration $S[1] \cdot S[2] \cdots S[k](I)$. Let $1 \leq k, k' \leq |S|$, $S[k] = (p, m, d)$ and $S[k'] = (p', m', d')$. We say that $S[k]$ *causally precedes* $S[k']$ *in* R, and we write $S[k] \rightarrow_R S[k']$, if one of the following conditions hold:

(1) $p = p'$ and $k \leq k'$;

(2) $m' \neq \lambda$ and p *sends* m' in $S[k]$, i.e., $m' \in M[k] - M[k-1]$;

(3) $\exists k'$, $1 \leq k'' \leq k'$: $S[k] \rightarrow_R S[k'']$ and $S[k''] \rightarrow_R S[k']$.

It follows from the definition of causality that if, for some processes p and q, the last step of p in a partial run R is not causally preceded by any step of q, then, no matter how we modify the initial state of q in R, p will not notice it. Formally:

Lemma 2.3 *Let $\mathcal{A}$ be any algorithm, $R = \langle F, H, I, S, T\rangle$ be any partial run of $\mathcal{A}$, and p, q be any processes such that no step of q causally precedes the last step of p in R. Then for any initial configuration I' of $\mathcal{A}$ such that I and I' differ only in the state of q, there exists a partial run $R' = \langle F, H, I', S', T\rangle$ of $\mathcal{A}$ such that p has the same state in $S(I)$ and $S'(I')$.*

2.10 Reducibility

Let $\mathcal{D}$ and $\mathcal{D}'$ be failure detectors, and $\mathcal{E}$ be an environment. If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is an algorithm $T_{\mathcal{D}' \rightarrow \mathcal{D}}$ that transforms $\mathcal{D}'$ into $\mathcal{D}$ in $\mathcal{E}$, we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$, and we write $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$.

If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ but $\mathcal{D}' \not\preceq_{\mathcal{E}} \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$, and we write $\mathcal{D} \prec_{\mathcal{E}} \mathcal{D}'$. If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ and $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$, we say that $\mathcal{D}$ *and* $\mathcal{D}'$ *are equivalent in* $\mathcal{E}$. If $\mathcal{D} \not\preceq_{\mathcal{E}} \mathcal{D}'$ and $\mathcal{D}' \not\preceq_{\mathcal{E}} \mathcal{D}$, we say that $\mathcal{D}$ *and* $\mathcal{D}'$ *are incomparable in* $\mathcal{E}$.

Algorithm $T_{\mathcal{D}' \rightarrow \mathcal{D}}$ that emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$ is called a *reduction algorithm.* Note that $T_{\mathcal{D}' \rightarrow \mathcal{D}}$ does not need to emulate *all* histories of $\mathcal{D}$; it is required that all the histories it emulates be histories of $\mathcal{D}$.

[2] For any sequences u and w, $u \cdot w$ denotes the concatenation of the two sequences.

2.11 A weakest failure detector

We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ if the following conditions are satisfied:

(a) $\mathcal{D}$ is *sufficient* to solve $\mathcal{M}$ in $\mathcal{E}$, i.e., $\mathcal{D}$ solves $\mathcal{M}$ in $\mathcal{E}$, and

(b) $\mathcal{D}$ is *necessary* to solve $\mathcal{M}$ in $\mathcal{E}$, i.e., if a failure detector $\mathcal{D}'$ solves $\mathcal{M}$ in $\mathcal{E}$, then $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$.

There might be a number of distinct failure detectors satisfying these conditions. (Though all such failure detectors are in the just defined sense equivalent.) Hence, it would be more technically correct to talk about *a* weakest failure detector to solve $\mathcal{M}$ in $\mathcal{E}$.

Chapter 3

Background

The first "weakest failure detector" result, referred here as the CHT proof, appeared in the seminal paper by Chandra, Hadzilacos and Toueg [14]. They showed that Ω is the weakest failure detector for solving consensus in asynchronous message-passing systems with a majority of correct processes.

Another milestone in the "weakest failure detector" research trend was determining the weakest failure detector for implementing read/write shared memory [20]. Combined with earlier work on state machine replication [50, 66], this result leads to determining the weakest failure detectors for solving consensus in *all* environments, i.e., for all assumptions on when and where failures might occur.

The approach taken in [14] and the result of [20] inspired some of the techniques derived in this thesis. Section 3.1 gives a short overview of the technical details of the approach, and Section 3.2 shows how the approach can be used to derive the result of [20]. Both results are important to recall in order to comprehend the results of this thesis.

3.1 The CHT proof

Let $\mathcal{E}$ be any environment, $\mathcal{D}$ be any failure detector that can be used to solve consensus in $\mathcal{E}$, and $\mathcal{A}$ be any algorithm that solves consensus in $\mathcal{E}$ using $\mathcal{D}$. We determine a reduction algorithm $T_{\mathcal{D}\rightarrow\Omega}$ that, using failure detector $\mathcal{D}$ and algorithm $\mathcal{A}$, implements Ω in $\mathcal{E}$. Recall that implementing Ω means outputting, at every process, the id of a process so that eventually, the id of the same correct process is output permanently at all correct processes.

3.1.1 Overview of the reduction algorithm

The basic idea underlying $T_{\mathcal{D}\rightarrow\Omega}$ is to have each process locally *simulate* the overall distributed system in which the processes execute several runs of $\mathcal{A}$ that *could have happened* in the current failure pattern and failure detector history. Every process then uses these runs to extract Ω.

In the local simulations, every process p feeds algorithm $\mathcal{A}$ with a set of proposed values, one for each process of the system. Then all automata composing $\mathcal{A}$ are triggered locally by p which emulates, for every simulated run of $\mathcal{A}$, the

```
$G_p \leftarrow$ empty graph
$k_p \leftarrow 0$
while true do
    receive message $m$
    $d_p \leftarrow$ query failure detector $\mathcal{D}$
    $k_p \leftarrow k_p + 1$
    if $m$ is of the form $(q, G_q, p)$ then $G_p \leftarrow G_p \cup G_q$
    add $[p, d_p, k_p]$ and edges from all vertices of $G_p$ to $[p, d_p, k_p]$ to $G_p$
    send $(p, G_p, q)$ to all $q \in \Pi$
```

Figure 3.1: Building a DAG: process p

states of all processes as well as the emulated buffer of exchanged messages.

Crucial elements that are needed for the simulation are (1) the values from failure detectors that would be output by $\mathcal{D}$ as well as (2) the order according to which the processes are taking steps. For these elements, which we call the stimuli of algorithm $\mathcal{A}$, process p periodically queries its failure detector module and exchanges the failure detector information with the other processes.

The reduction algorithm $T_{\mathcal{D}\to\Omega}$ consists of two tasks that are run in parallel at every process: the *commmuncation task* and the *computation task*. In the communication task, every process maintains ever-growing stimuli of algorithm $\mathcal{A}$ by periodically querying its failure detector module and sending the output to all other processes. In the computation task, every process periodically feeds the stimuli to algorithm $\mathcal{A}$, simulates several runs of $\mathcal{A}$, and computes the current emulated output of Ω.

3.1.2 Building a DAG

The communication task of algorithm $T_{\mathcal{D}\to\Omega}$ is presented in Figure 3.1. Executing this task, p knows more and more of the processes' failure detector outputs and temporal relations between them. All this information is pieced together in a single data structure, a directed acyclic graph (DAG) G_p. Informally, every vertex $[q, d, k]$ of G_p is a failure detector value "seen" by q in its k-th query of its failure detector module. An edge $([q, d, k], [q', d', k'])$ can be interpreted as "q saw failure detector value d (in its k-th query) *before* q' saw failure detector value d' (in its k'-th query)".

DAG G_p has some special properties which follow from its construction. Let F be the current failure pattern in $\mathcal{E}$ and H be the current failure detector history in $\mathcal{D}(F)$. Then:

(1) The vertices of G_p are of the form $[q, d, k]$ where $q \in \Pi$, $d \in \mathcal{R}_{\mathcal{D}}$ and $k \in \mathbb{N}$. There is a mapping $\tau :$ vertices of $G_p \mapsto \mathbb{T}$, associating a time with every vertex of G_p, such that:

   (a) For any vertex $v = [q, d, k]$, $q \notin F(\tau(v))$ and $d = H(q, \tau(v))$. That is, d is the value output by q's failure detector module at time $\tau(v)$.

   (b) For any edge (v, v') in G_p, $\tau(v) < \tau(v')$. That is, any edge in G_p reflects the temporal order in which the failure detector values are output.

(2) If $v' = [q, d, k]$ and $v'' = [q, d', k']$ are vertices of G_p, and $k < k'$, then (v, v') is an edge of G_p.

(3) G_p is transitively closed: if (v, v') and (v', v'') are edges of G_p, then (v, v'') is also an edge of G_p.

(4) For all correct processes p and q and all times t, there is a time $t' \geq t$, a $d \in \mathcal{R}_{\mathcal{D}}$ and a $k \in \mathbb{N}$ such that for every vertex v of $G_p(t)$, $(v, [q, d, k])$ is an edge of $G_p(t')$.[1]

Note that properties (1)–(4) imply that, for every correct process p, $t \in \mathbb{T}$ and $k \in \mathbb{N}$, there is a time t' such that $G_p(t')$ contains a path $g = [q_1, d_1, k_1] \rightarrow [q_2, d_2, k_2] \rightarrow \ldots$, such that (a) every correct process appears at least k times in g, and (b) for any path g' in $G_p(t)$, $g' \cdot g$ is also a path in $G_p(t')$.

3.1.3 Simulation trees

Now DAG G_p can be used to simulate runs of $\mathcal{A}$. Any path $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots, [q_s, d_s, k_s]$ through G_p gives the order in which processes $q_1, q_2, \ldots, q_s$ "see", respectively, failure detector values $d_1, d_1, d_2, \ldots, d_s$. That is, g contains an activation schedule and failure detector outputs for the processes to execute steps of $\mathcal{A}$'s instances. Let I be any initial configuration of $\mathcal{A}$. Consider a schedule S that is applicable to I and *compatible with* g, i.e., $|S| = s$ and $\forall k \in \{1, 2, \ldots, s\}$, $S[k] = (q_k, m_k, d_k)$, where m_k is a message addressed to q_k (or the null message λ).

All schedules that are applicable to I and compatible with paths in G_p can be represented as a tree Υ_p^I, called the *simulation tree induced by* G_p *and* I. The set of vertices of Υ_p^I is the set of all schedules S that are applicable to I and compatible with paths in G_p. The root of Υ_p^I is the empty schedule $S_\perp$. There is an edge from S to S' if and only if $S' = S \cdot e$ for a step e; the edge is labeled e. Thus, every vertex S of Υ_p^I is associated with a sequence of steps $e_1\, e_2 \ldots e_s$ consisting of labels of the edges on the path from $S_\perp$ to S. In addition, every descendant of S in Υ_p^I corresponds to an extension of $e_1\, e_2 \ldots e_s$.

The construction of Υ_p^I implies that, for any vertex S of Υ_p^I, there exists a partial run $\langle F, H, I, S, T \rangle$ of $\mathcal{A}$ where F is the current failure pattern and $H \in \mathcal{D}(F)$ is the current failure detector history. Thus, if in S, correct processes appear sufficiently often and receive sufficiently many messages sent to them, then every correct (in F) process decides in $S(I)$.

In the example depicted in Figure 3.2, a DAG (a) induces a simulation tree a portion of which is shown in (b). There are three non-trivial paths in the DAG: $[p_1, d_1, k_1] \rightarrow [p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$ and $[p_1, d_1, k_1] \rightarrow [p_1, d_3, k_3]$. Every path through the DAG and an initial configuration I induce at least one schedule in the simulation tree. Hence, the simulation tree has at least three leaves: (p_1, λ, d_1) (p_2, m_2, d_2) (p_1, m_3, d_3), (p_2, λ, d_2) (p_1, m_3', d_3), and (p_1, λ, d_3). Recall that λ is the empty message: since the message buffer is empty in I, no non-empty message can be received in the first step of any schedule.

[1] For any variable x and time t, $x(t)$ denotes the value of x at time t.

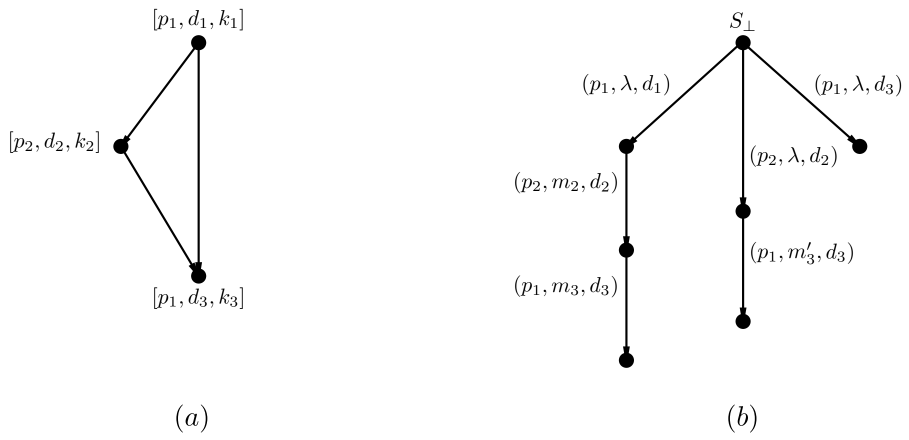

Figure 3.2: A DAG and a tree

3.1.4 Tags and valences

Let I^i, $i \in \{0, 1, \ldots, n\}$ denote the initial configuration of $\mathcal{A}$ in which processes $p_1, \ldots, p_i$ propose 1 and the rest (processes $p_{i+1}, \ldots, p_n$) propose 0. In the computation task of the reduction algorithm, every process p maintains an ever-growing *simulation forest* $\Upsilon_p = \{\Upsilon_p^0, \Upsilon_p^1, \ldots, \Upsilon_p^n\}$ where Υ_p^i $(0 \leq i \leq n)$ denotes the simulation trees induced by G_p and initial configurations I^i.

For every vertex of the simulation forest, p assigns a set of *tags*. Vertex S of tree Υ_p^i is assigned a tag v if and only if S has a descendant S' in Υ_p^i such that p decides v in $S'(I^i)$. We call the set tags the *valence* of the vertex. By definition, if S has a descendant with a tag v, then S has tag v. Validity of consensus ensures that the set of tags is a subset of $\{0, 1\}$.

Of course, at a given time, some vertices of the simulation forest Υ_p might not have any tags because the simulation stimuli are not sufficiently long yet. But this is just a matter of time: if p is correct, then every vertex of p's simulation forest will eventually have an extension in which correct processes appear sufficiently often for p to take a decision.

A vertex S of Υ_p^i is 0-*valent* if it has exactly one tag $\{0\}$ (only 0 can be decided in S's extensions in Υ_p^i). A 1-valent vertex is analogously defined. If a vertex S has both tags 0 and 1 (both 0 and 1 can be decided in S's extensions), then we say that S is *bivalent*.[2]

It immediately follows from Validity of consensus that the root of Υ_p^0 can at most be 0-valent, and the root of Υ_p^n can at most be 1-valent (the roots of Υ_p^0 and Υ_p^n cannot be bivalent).

3.1.5 Stabilization

Note that the simulation trees can only grow with time. As a result, once a vertex of the simulation forest Υ_p gets a tag v, it cannot lose it later. Thus, eventually

[2] The notion of valence was first defined in [28] as the set of values than are decided in *all* extensions of a given execution. Here we define the valence as only a subset of these values, defined by the simulation tree.

every vertex of Υ_p stabilizes being 0-valent, 1-valent, or bivalent. Since correct processes keep continuously exchanging the failure detector samples and updating their simulation forests, every simulation tree computed by a correct process at any given time will eventually be a subtree of the simulation forest of every correct process.

Formally, let p be any correct process, t be any time, i be any index in $\{0, 1, \dots, n\}$, and S be any vertex of $\Upsilon_p^i(t)$. Then:

(i) There exists a non-empty $V \subseteq \{0, 1\}$ such that there is a time after which the valence of S is V. (We say that the valence of S *stabilizes* on V at p.)

(ii) If the valence of S stabilizes on V at p, then for every correct process q, there is a time after which S is a vertex of Υ_q^i and the valence of S stabilizes on V at q.

Hence, the correct processes eventually agree on the same tagged simulation subtrees. In discussing the stabilized tagged simulation forest, it is thus convenient to consider the *limit* infinite DAG G and the *limit* infinite simulation forest $\Upsilon = \{\Upsilon^0, \Upsilon^1, \dots, \Upsilon^n\}$ such that for all $i \in \{0, 1, \dots, n\}$ and all correct processes p, $\cup_{t \in \mathbb{T}} G_p(t) = G$ and $\cup_{t \in \mathbb{T}} \Upsilon_p^i(t) = \Upsilon^i$.

3.1.6 Critical index

Let p be any correct process. We say that index $i \in \{1, 2, \dots, n\}$ is *critical* if *either* the root of Υ^i is bivalent *or* the root of Υ^{i-1} is 0-valent and the root of Υ^i is 1-valent. In the first case, we say that i is *bivalent critical.* In the second case, we say that i is *univalent critical.*

Lemma 3.1 *There is at least one critical index in* $\{1, 2, \dots, n\}$.

Proof: Indeed, by the Validity property of consensus, the root of Υ^0 is 0-valent, and the root of Υ^1 is 1-valent. Thus, there must be an index $i \in \{1, 2, \dots, n\}$ such that the root of Υ^{i-1} is 0-valent, and Υ^i is either 1-valent or bivalent. □

Since tagged simulation forests computed at the correct processes tend to the same infinite tagged simulation forest, eventually, all correct processes compute the same *smallest* critical index i of the same type (univalent or bivalent). Now we have two cases to consider for the smallest critical index: (1) i is univalent critical, or (2) i is bivalent critical.

(1) Handling univalent critical index

Lemma 3.2 *If i is univalent critical, then p_i is correct.*

Proof: By contradiction, assume that p_i is faulty. Then G contains an infinite path g in which p_i does not participate and every correct process participates infinitely often. Then Υ^i contains a vertex S such that p_i does not take steps in S and some correct process p decides in $S(I^i)$. Since i is 1-valent, p decides 1 in $S(I^i)$. But p_i is the only process that has different states in I^{i-1} and I^i, and p_i does not take part in S. Thus, S is also a vertex of Υ^{i-1} and p decides 1 in

$S(I^{i-1})$. But the root of Υ^{i-1} is 0-valent — a contradiction. □

(2) Handling bivalent critical index

Assume now that the root of Υ^i is *bivalent.* Below we show that Υ^i then contains a *decision gadget*, i.e., a finite subtree which is either a *fork* or a *hook* (Figure 3.3).

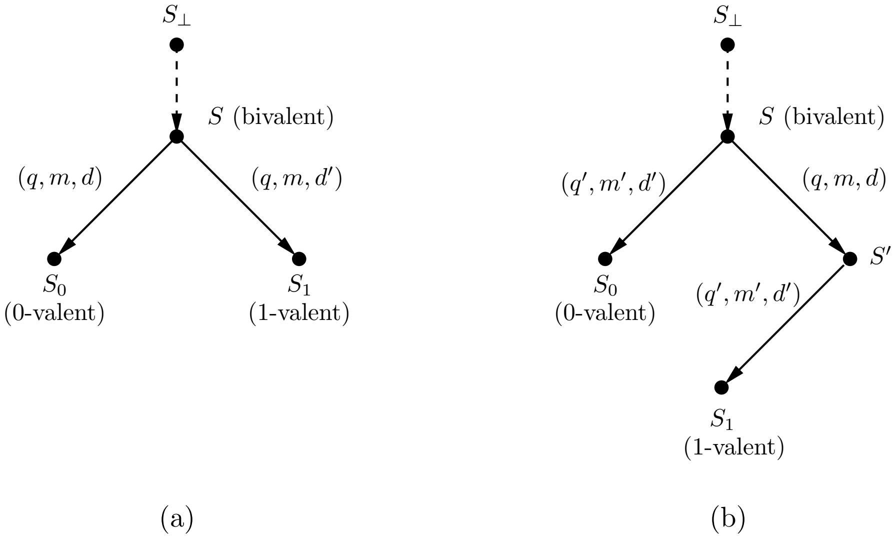

Figure 3.3: A fork and a hook

A fork (case (a) in Figure 3.3) consists of a bivalent vertex S from which two *different* steps by the *same* process q, consuming the same message m, are possible which lead, on the one hand, to a 0-valent vertex S_0 and, on the other hand, to a 1-valent vertex S_1.

A hook (case (b) in Figure 3.3) consists of a bivalent vertex S, a vertex S' which is reached by executing a step of some process q, and two vertices S_0 and S_1 reached by applying *the same* step of process q' to, respectively, S and S'. Additionally, S_0 must be 0-valent and S_1 must be 1-valent (or vice versa; the order does not matter here).

In both cases, we say that q is the *deciding process*, and S is the *pivot* of the decision gadget.

Lemma 3.3 *The deciding process of a decision gadget is correct.*

Proof: Consider any decision gadget γ defined by a pivot S, vertices S_0 and S_1 of opposite valence and a deciding process q. By contradiction, assume that q is faulty. Let g, g_0 and g_1 be the simulation stimuli of, respectively, S, S_0 and S_1. Then G contains an infinite path $\tilde{g}$ such that (a) $g \cdot \tilde{g}$, $g_0 \cdot \tilde{g}$, $g_1 \cdot \tilde{g}$ are paths in G, and (b) q does not appear and the correct processes appear infinitely often in g.

Let γ be a fork (case (a) in Figure 3.3). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of $\tilde{g}$ and applicable to $S(I^i)$ such that some correct process p decides in $S \cdot \tilde{S}(I^i)$; without loss of generality, assume that p decides 0. Since q is the only process that can distinguish $S(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to $S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$. But S_1 is 1-valent — a contradiction.

Let γ be a hook (case (b) in Figure 3.3). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of g and applicable to $S_0(I^i)$ such that some correct process p decides in $S_0 \cdot \tilde{S}(I^i)$. Without loss of generality, assume that S_0 is 0-valent, and hence p decides 0 in $S_0 \cdot \tilde{S}(I^i)$. Since q is the only process that can distinguish $S_0(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to $S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$ But S_1 is 1-valent — a contradiction. □

Now we need to show that any bivalent simulation tree Υ^i contains at least one decision gadget γ.

Lemma 3.4 *If i is bivalent critical, then Υ^i contains a decision gadget.*

Proof: Let i be a bivalent critical index. In Figure 3.4, we present a procedure which goes through Υ^i. The algorithm starts from the bivalent root of Υ^i and terminates when a hook or a fork has been found.

```
$S \leftarrow S_\perp$
while $true$ do
   $p \leftarrow$ ⟨choose the next correct process in a round robin fashion⟩
   $m \leftarrow$ ⟨choose the oldest undelivered message addressed to $p$ in $S(I^i)$⟩
   if ⟨$S$ has a descendant $S'$ in $\Upsilon^i$ (possibly $S = S'$) such that, for some $d$,
         $S' \cdot (p, m, d)$ is a bivalent vertex of $\Upsilon^i$⟩
   then $S \leftarrow S' \cdot (p, m, d)$
   else exit
```

Figure 3.4: Locating a decision gadget

We show that the algorithm indeed terminates. Suppose not. Then the algorithm locates an infinite *fair path* through the simulation tree, i.e., a path in which all correct processes get scheduled infinitely often and every message sent to a correct process is eventually consumed. Additionally, this fair path goes through bivalent states only. But no correct process can decide in a bivalent state $S(I^i)$ (otherwise we would violate the Agreement property of consensus). As a result, we constructed a run of $\mathcal{A}$ in which no correct process ever decides — a contradiction.

Thus, the algorithm in Figure 3.4 terminates. That is, there exist a bivalent vertex S, a correct process p, and a message m addressed to p in $S(I^i)$ such that

(*) For all descendants S' of S (including $S' = S$) and all d, $S' \cdot (p, m, d)$ is *not* a bivalent vertex of Υ^i.

In other words, any step of p consuming message m brings any descendant of S (including S itself) to either a 1-valent or a 0-valent state. Without loss of generality, assume that, for some d, $S \cdot (p, m, d)$ is a 0-valent vertex of Υ^i. Since S is bivalent, it must have a 1-valent descendant S''.

If S'' includes a step in which p consumes m, then we define S' as the vertex of Υ^i such that, for some d', $S' \cdot (p, m, d')$ is a prefix of S''. If S'' includes no step in which p consumes m, then we define $S' = S''$. Since p is correct, for some d', $S' \cdot (p, m, d')$ is a vertex of Υ^i. In both cases, we obtain S' such that for some d', $S' \cdot (p, m, d')$ is a 1-valent vertex of Υ^i.

Let the path from S to S' go through the vertices $\sigma_0 = S, \sigma_1, \ldots, \sigma_{m-1}, \sigma_m = S'$. By transitivity of G, for all $k \in \{0, 1, \ldots, m\}$, $\sigma_k \cdot (p, m, d')$ is a vertex of Υ^i. By (*), $\sigma_k \cdot (p, m, d')$ is either 0-valent or 1-valent vertex of Υ^i.

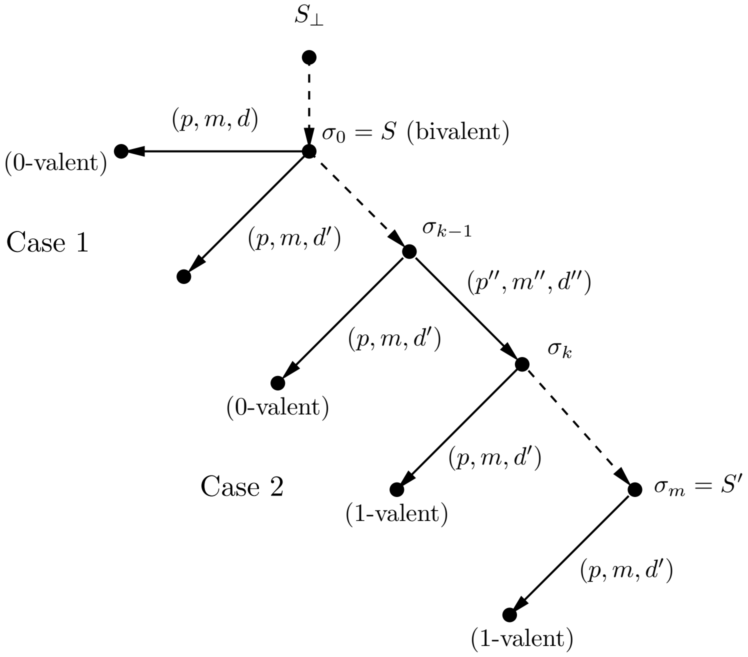

Figure 3.5: Locating a fork (Case 1) or a hook (Case 2)

Let $k \in \{0, \ldots, m\}$ be the lowest index such that (p, m, d') brings σ_k to a 1-valent state. We know that such an index exists, since $\sigma_m \cdot (p, m, d')$ is 1-valent and all such resulting states are either 0-valent or 1-valent.

Now we have the following two cases to consider: (1) $k = 0$, and (2) $k > 0$.

Assume that $k = 0$, i.e., (p, m, d') applied to S brings it to a 1-valent state. But we know that there is a step (p, m, d) that brings S to a 0-valent state (Case 1 in Figure 3.5). That is, a fork is located!

If $k > 0$, we have the following situation. Step (p, m, d') brings σ_{k-1} to a 0-valent state, and $\sigma_k = \sigma_{k-1} \cdot (p', m', d'')$ to a 1-valent state (Case 2 in Figure 3.5). But that is a hook!

As a result, any bivalent infinite simulation tree has at least one decision gadget. □

3.1.7 The reduction algorithm

Now we are ready to complete the description of $T_{\mathcal{D}\to\Omega}$. In the computation task (Figure 3.6), every process p periodically extracts the current *leader* from its simulation forest, so that eventually the correct processes agree on the same correct leader. The current leader is stored in variable Ω-*output*$_p$.

```
Initially:
   for $i = 0, 1, \ldots, n$: $\Upsilon_p^i \leftarrow$ empty graph
   $\Omega$-$output_p \leftarrow p$

while true do

   { Build and tag the simulation forest induced by $G_p$ }
   for $i = 0, 1, \ldots, n$ do
      $\Upsilon_p^i \leftarrow$ simulation tree induced by $G_p$ and $I^i$
      for every vertex $S$ of $\Upsilon_p^i$:
         if $S$ has a descendant $S'$ such that $p$ decides $v$ in $S'(I^i)$ then
            add tag $v$ to $S$

   { Select a process from the tagged simulation forest }
   if there is a critical index then
      $i \leftarrow$ the smallest critical index
      if $i$ is univalent critical then $\Omega$-$output_p \leftarrow p_i$
      if $\Upsilon_p^i$ has a decision gadget then
         $\Omega$-$output_p \leftarrow$ the deciding process of the smallest decision gadget in $\Upsilon_p^i$
```

Figure 3.6: Extracting a correct leader: code for each process p

Initially, p elects itself as a leader. Periodically, p updates its simulation forest Υ_p by incorporating more simulation stimuli from G_p. If the forest has a univalent critical index i, then p outputs p_i as the current leader estimate. If the forest has a bivalent critical index i and Υ_p^i contains a decision gadget, then p outputs the deciding process of *the smallest* decision gadget in Υ_p^i (the "smallest" can be well-defined, since the vertices of the simulation tree are countable).

Eventually, the correct processes locate the same *stable* critical index i. Now we have two cases to consider:

(i) i is univalent critical. By Lemma 3.2, p_i is correct.

(ii) i is bivalent critical. By Lemma 3.4, the limit simulation tree Υ^i contains a decision gadget. Eventually, the correct processes locate the same decision gadget γ in Υ_i and compute the deciding process q of γ. By Lemma 3.3, q is correct.

Thus, eventually, the correct processes elect the same correct leader — Ω is emulated!

3.1.8 Multivalent critical index

We can easily adapt the notion of valence to a *multivalued* version of consensus where processes agree on a value in a set V of two or more elements.

Let I be any initial configuration of a multivalued consensus algorithm $\mathcal{A}$. Again, let Υ_p^I denote the tagged simulation tree induced by G_p and I. We say that a vertex S of Υ_p^I is *u-valent* if it has only one tag u ($u \in V$). We say that a vertex S of Υ_p^I is *multivalent* if it has two or more tags.

Let $I^0, I^1, \ldots, I^m$ be initial configurations of $\mathcal{A}$ such that for all $i = 1, 2, \ldots, m$, I^{i-1} and I^i differ in the state of exactly one process. Again, let Υ_p^i denote the tagged simulation tree induced by G_p and I^i, $i = 0, 1, \ldots, m$. We define i to be *critical* if the root of Υ^i is multivalent (in which case i is called *multivalent critical*), or if the root of Υ^{i-1} is u-valent and the root of Υ^i is v-valent, where $u, v \in V$, $u \neq v$, and I^{i-1} and I^i differ in the state of one process (in which case i is called *univalent critical*). Note that the notions of decision gadgets as well as Lemmas 3.3 and 3.4 can be easily generalized to handle multivalent critical indices. We will come back to this issue in Section 5.2.

3.1.9 Weak consensus vs. consensus

In *weak consensus*, the Validity property of consensus is replaced with the following weaker property:

Non-Triviality: Every algorithm that solves weak consensus must have a run in which 0 is decided by some process, and a run in which 1 is decided by some process.

The proof of [14] does not show that Ω is necessary to solve *weak consensus.* More precisely, Lemma 3.1 does not hold anymore: it might happen that, in a given execution of the reduction algorithm, all trees of the simulation forest Υ are, say, 0-valent. Thus, there might be no critical index, i.e., the failure detector history does not help to extract Ω.

In fact, Ω is not the weakest failure detector for solving weak consensus. The failure detector $\mathcal{D}$ that, for any failure pattern, can output either 0 at all processes, or 1 at all processes, is sufficient to solve weak consensus in any environment [14]. It is straightforward to show that Ω is not weaker than $\mathcal{D}$ in any nontrivial environment (e.g., $\mathcal{E}_f$ with $0 < f < n$).

In a sense, there is a gap between the weak consensus problem considered in the impossibility proof of Fischer, Lynch and Paterson [28], and the consensus problem for which the weakest failure detector was determined by Chandra, Hadzilacos and Toueg [14]. Interestingly, the gap collapses when the consensus impossibility is circumvented using *deterministic* shared objects with sequential specifications: any objects that can be used to solve weak consensus, are powerful enough to solve consensus [36].

3.2 Implementing a register

In this section, we show that the quorum failure detector Σ is the weakest failure detector to implement atomic registers in all environments. The result was first obtained by Delporte-Gallet, Fauconnier and Guerraoui [20]. An alternative “CHT-like” proof, based on exchanging failure detector samples and using the samples as stimuli for locally simulated runs, was later presented in [26]. We review here the proof of [26], because a similar technique is employed in this thesis to determine the weakest failure detector for solving quittable consensus (Section 5.2).

3.2.1 Read/write shared memory

A *register* is a shared object accessed through two operations: *read* and *write*. The write operation takes as an input parameter a specific value to be stored in the register and returns a simple indication *ok* that the operation has been executed. The read operation takes no parameters and returns a value according to one of the following consistency criteria. A (single-writer, multi-reader) *safe* register ensures only that any read operation that does not overlap with any other operation returns the argument of the last write operation. A stronger *regular* ensures that any read operation returns either a concurrently written value, or the value written by the last write operation. The strongest *atomic* register ensures that any operation appears to be executed instantaneously between its invocation and reply time events. (Precise definitions are given in [43, 9].)

The registers we consider are *fault-tolerant*: they ensure that, despite concurrent invocations and possible crashes of the processes, every correct process that invokes an operation eventually gets a reply (a value for the read and an *ok* indication for the write).

The classical results [71, 44] imply that if a failure detector $\mathcal{D}$ is sufficient to implement a safe one-writer one-reader register for any two processes, then $\mathcal{D}$ is sufficient to implement an atomic multi-writer multi-reader register. Thus, we do not need to specify here whether the register implemented using $\mathcal{D}$ is safe, regular or atomic: all these registers are computationally equivalent.

3.2.2 The sufficiency part

Recall that the quorum failure detector Σ outputs a set of processes at each process. Any two sets (output at any times and by any processes) intersect, and eventually every set consists of only correct processes.

By a simple variation of the algorithm of Attiya *et al.* for implementing registers in a message-passing system with a majority of correct processes [8], we obtain an algorithm that implements registers in any environment using Σ. Where the original algorithm uses waiting until a majority responds to ensure that a read operation returns the most recently written value, we can use the quorums provided by Σ to the same effect.

3.2.3 The reduction algorithm

Now we need to show that any failure detector that can be used to implement registers can be transformed into Σ.

Let $\mathcal{E}$ be any environment. Let $\mathcal{D}$ be any failure detector that can be used to implement in $\mathcal{E}$ a set of atomic registers $\{X_p\}_{p\in\Pi}$, where for every $p \in \Pi$, X_p can be written by p and read by all processes. We present an algorithm that, using $\mathcal{D}$, implements Σ.

To extract Σ, we assign a particular protocol, i.e., a sequence of operations on the implemented registers, to every process. In this protocol, denoted $\mathcal{A}$, every

```
Initially:
  $\Sigma$-$output_p \leftarrow \Pi$ { $\Sigma$-$output_p$ is the output of $p$'s module of $\Sigma$ }
  $I \leftarrow$ the initial configuration of $\mathcal{A}$

while true do
  wait until $p$ adds a new failure detector sample
    $u$ to its DAG $G_p$
  repeat
    let $G_p(u)$ be the subgraph induced by
      the descendants of $u$ in $G_p$
    $\mathcal{S} \leftarrow$ set of all schedules of $\mathcal{A}$
      compatible with some path of $G_p(u)$ and applicable to $I$
  until there is a schedule $S \in \mathcal{S}$ of a complete $p$-solo run of $\mathcal{A}$
  $\Sigma$-$output_p \leftarrow$ set of all processes that take steps in the schedule $S$
```

Figure 3.7: Extracting Σ: code for each process p

process p first writes 1 in X_p, and then reads the registers $\{X_q\}_{q\in\Pi}$ (we assume that each X_q is initialized to 0). A run in which p is the only process that executes $\mathcal{A}$, is called a *p-solo run* of $\mathcal{A}$. A p-solo run in which p completes $\mathcal{A}$, is called a *complete p-solo run* of $\mathcal{A}$.

It is important to notice that in any run R of $\mathcal{A}$ in which two processes p and q both complete executing $\mathcal{A}$, either p reads 1 in X_q, or q reads 1 in X_p. Intuitively, this implies that the sets of processes "involved" in the executions of $\mathcal{A}$ at p and q intersect, which gives us a hint of how to extract Σ from $\mathcal{A}$ and $\mathcal{D}$.

Again, the reduction algorithm consists of two tasks: the communication task and the computation task.

The communication task, in which each process p samples its local module of $\mathcal{D}$, exchanges the failure detector samples with the other processes, and assembles these samples in an ever-increasing directed acyclic graph G_p, is organized exactly as in Section 3.1 (Figure 3.1). The computation task, in which p simulates runs of $\mathcal{A}$ and uses these runs to extract its current *quorum* (the output of its emulated module of Σ), is presented in Figure 3.7.

To compute its current quorum, process p first waits until enough "fresh" (not previously appeared) failure detector samples are collected in G_p. Eventually, G_p includes a sufficiently long fresh path g such that there is a schedule S of a complete p-solo run of $\mathcal{A}$, compatible with g. The set of processes that take steps in S constitute the current quorum of p stored in variable Σ-$output_p$.

The correctness of the reduction algorithm follows immediately from the following two observations:

(1) Eventually, at every correct process p, Σ-$output_p$ contains only correct processes.

    Indeed, there is a time after which faulty processes do not produce fresh failure detector samples and thus do not participate in fresh schedules of $\mathcal{A}$ simulated by p.

(2) For all p and q, Σ-$output_p$ and Σ-$output_q$ always intersect.

> Indeed, assume, by contradiction, that there exist $P, Q \subset \Pi$ such that $P \cap Q = \emptyset$, and, at some time t_1, p computes $\Sigma\text{-}output_p = P$ and, at some time t_2, q computes $\Sigma\text{-}output_q = Q$.
>
> By the algorithm of Figure 3.7, $\mathcal{A}$ has a complete p-solo run $R_p = \langle F, H, I, S_p, T_p \rangle$ and a complete q-solo run $R_q = \langle F, H, I, S_q, T_q \rangle$ such that the sets of processes that participate in S_p and S_q are disjoint.
>
> By Lemma 2.1, the two runs R_p and R_q can be composed in a single run $R = \langle F, H, I, S, T \rangle$ that is indistinguishable from R_p to p, and indistinguishable from R_q to q.
>
> Hence, both p and q complete $\mathcal{A}$ in R. Since R_p is a p-solo run, and p cannot distinguish R and R_p, p reads 0 from register X_q in R. Respectively, q reads 0 from register X_p in R. But this cannot happen in any register implementation: at least one of the processes p and q must read 1 in the register of the other process!
>
> The contradiction implies that $\Sigma\text{-}output_p$ and $\Sigma\text{-}output_q$ always intersect.

Thus, for all environments $\mathcal{E}$, Σ is the weakest failure detector to implement atomic registers in $\mathcal{E}$.

3.2.4 Solving consensus in all environments

Once we determined the weakest failure detector to implement atomic registers, determining the weakest failure detector for solving consensus in all environments is straightforward. This failure detector is (Ω, Σ), the composition of Ω and Σ.

Indeed, failure detector (Ω, Σ) can be used to solve consensus in all environment, by first implementing registers out of Σ, and then consensus out of registers and Ω [53].

On the other hand, consensus can be used to implement atomic registers in any environment [50, 66], and thus to extract Σ. Combined with the fact that Ω is necessary to solve consensus in *any* environment [14] (see Section 3.1), this implies that (Ω, Σ) is necessary to solve consensus in any environment.

Chapter 4

Mutual Exclusion

This chapter considers the *fault-tolerant mutual exclusion* problem (FTME) in a message-passing asynchronous system and determines the weakest failure detector to solve the problem, given a majority of correct processes. This failure detector, which we call the *trusting* failure detector, and which we denote by $\mathcal{T}$, is strictly weaker than the perfect failure detector $\mathcal{P}$ but strictly stronger than the eventually perfect failure detector $\diamond\mathcal{P}$. We show that a majority of correct processes is necessary to solve FTME with $\mathcal{T}$. Moreover, $\mathcal{T}$ is also the weakest failure detector to solve the fault-tolerant *group* mutual exclusion problem (FTGME), given a majority of correct processes.

Section 4.1 defines the fault-tolerant mutual exclusion problem. Section 4.2 introduces the trusting failure detector $\mathcal{T}$. Sections 4.3 and 4.4 show that $\mathcal{T}$ is, respectively, necessary and sufficient to solve the problem. Section 4.5 discusses the bounds on the number of correct processes necessary to solve the problem with $\mathcal{T}$ and introduces a failure detector $(\mathcal{T}, \Sigma)$ which is sufficient to solve the problem without a majority of correct processes. Section 4.6 generalizes these results to the group mutual exclusion problem. Section 4.7 discusses the costs of resilience provided by $\mathcal{T}$. Section 4.8 discusses practical issues of implementing $\mathcal{T}$. Section 4.10 reviews the related work.

The results of this chapter appeared originally in [22].

4.1 The fault-tolerant mutual exclusion problem

FTME involves the allocation of a single, indivisible, resource among n processes. An alive (not crashed) process with access to the resource is said to be in its *critical section* (*CS*). When a process is not involved in any way with the resource, it is said to be in its *remainder section.* To gain access to its critical section, a process executes a *trying protocol*, and after the process is done with the resource, it executes an *exit protocol.* This procedure can be repeated, so each process i cyclically moves from its remainder section (rem_i) to its *trying section* (try_i), then to its critical section (crit_i), then to its *exit section* (exit_i), and then back again to rem_i. We assume that every process i is *well-formed*, i.e., i does not violate the cyclic order of execution: $\mathsf{rem}_i, \mathsf{try}_i, \mathsf{crit}_i, \mathsf{exit}_i, \ldots$.

A mutual exclusion algorithm defines trying protocol try_i and exit protocol

exit_i for every process i. (We do not restrict the process behavior in the critical and remainder sections.)

We say that the algorithm solves the FTME problem if, under the assumption that every process is well-formed, any run of the algorithm satisfies the following properties:

Mutual exclusion: No two different processes are in their CSs at the same time.

Progress:

(1) If a correct process is in its trying section, then at some time later some correct process is in its CS.

(2) If a correct process is in its exit section, then at some time later it enters its remainder section.

We will show in Sections 4.3 and 4.4 that, in any environment with a majority of correct processes, any algorithm that solves the FTME problem can be transformed into an algorithm satisfying not only the properties above but also the following "fairness" property:

Starvation freedom: If no process stays forever in its CS, then every correct process that reaches its trying section eventually enters its CS.

Note that mutual exclusion is a *safety* property while progress and starvation freedom are *liveness* properties.

An alternative stronger definition of the problem can allow a process to be initially in its CS. It is straightforward to show that the perfect failure detector $\mathcal{P}$ is necessary for this problem. Instead, we follow the classical definitions of the problem (see, for example, [57, chapter 10]) in which the competition between processes for the critical section is "fair", since none of them can usurp the CS from the very beginning.

4.2 The trusting failure detector

This section introduces a new failure detector, denoted $\mathcal{T}$, that we call the *trusting* failure detector. The range of $\mathcal{T}$ is $\mathcal{R}_{\mathcal{T}} = 2^{\Pi}$. Let $H_{\mathcal{T}}$ be any history of $\mathcal{T}$. $H_{\mathcal{T}}(i,t)$ represents the set of processes that process i *suspects* (i.e., considers to have crashed) at time t. We say that process i *trusts process j at time t* if $j \notin H_{\mathcal{T}}(i,t)$.

For every failure pattern F, $\mathcal{T}(F)$ is defined by the set of *all* histories $H_{\mathcal{T}}$ that satisfy the following properties:

Strong completeness: eventually, every crashed process is permanently suspected by every correct process. That is:

$$\forall i \notin \mathit{correct}(F), \exists t: \; \forall t' > t, \forall j \in \mathit{correct}(F), i \in H_{\mathcal{T}}(j,t')$$

Eventual strong accuracy: eventually, no correct process is suspected by any correct process. That is:

$$\forall i \in \mathit{correct}(F), \exists t: \; \forall t' > t, \forall j \in \mathit{correct}(F), i \notin H_{\mathcal{T}}(j, t')$$

Trusting accuracy: every process j that is suspected by a process i after being trusted by i is crashed. That is:

$$\forall i, j, t < t' : j \notin H_{\mathcal{T}}(i, t) \wedge j \in H_{\mathcal{T}}(i, t') \;\Rightarrow\; j \in F(t')$$

Note that the first two properties are the same as in the definition of the eventually perfect failure detector $\diamond\mathcal{P}$ [15] (Section 2.2).

Figure 4.1 depicts a possible scenario of failure detection with $\mathcal{T}$. Let $\Pi = \{p_1, p_2, p_3, p_4\}$. Initially, the failure detector module at process p_1 outputs $\{p_2, p_3, p_4\}$: $H(p_1, t_1) = \{p_2, p_3, p_4\}$, i.e., process p_1 trusts only itself. At time $t_2 > t_1$, processes p_2 and p_3 also get trusted by process p_1: $H(p_1, t_2) = \{p_4\}$. Process p_3 crashes and at some time later is not trusted anymore by process p_1: $\forall t \geq t_3, H(p_1, t) = \{p_3, p_4\}$. Note that process p_1 never trusts process p_4.

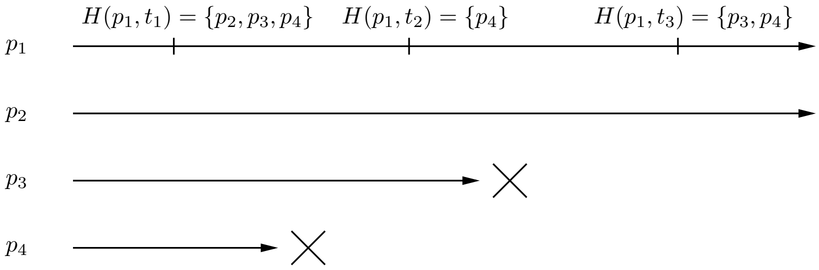
$H(p_1, t_1) = \{p_2, p_3, p_4\}$
$H(p_1, t_2) = \{p_4\}$
$H(p_1, t_3) = \{p_3, p_4\}$
p_1
p_2
p_3
p_4

Figure 4.1: Failure detection scenario for $\mathcal{T}$

Now we identify the position of $\mathcal{T}$ in the hierarchy of failure detectors introduced in [15]. We show that, in most environments, $\diamond\mathcal{P}$ is strictly weaker than $\mathcal{T}$, and $\mathcal{T}$ is strictly weaker than $\mathcal{P}$. The "weaker" parts of the proofs follow directly form the definition of $\mathcal{T}$. The "strictly" parts of the proofs are done by contradiction: we assume that a reduction algorithm $T_{\mathcal{T}\rightarrow\mathcal{P}}$ (respectively, $T_{\diamond\mathcal{P}\rightarrow\mathcal{T}}$) exists and expose a run of this reduction algorithm that violates some properties of $\mathcal{P}$ (respectively, $\mathcal{T}$).

Proposition 4.1 *Let $\mathcal{E}$ be any environment that contains a failure pattern in which some process initially crashes, and a failure-free failure pattern. Then $\mathcal{T} \prec_{\mathcal{E}} \mathcal{P}$.*

Proof:
(a) By definition, $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{P}$ in any environment $\mathcal{E}$: $\mathcal{P}$ satisfies all properties of $\mathcal{T}$. Indeed, strong completeness is given for free, eventual strong accuracy is implied by strong accuracy of $\mathcal{P}$. Trusting accuracy follows from the fact that $\mathcal{P}$ guarantees that any suspected process is crashed.

(b) Now we show that $\mathcal{P}$ is not weaker than $\mathcal{T}$. Intuitively, this follows from the fact that $\mathcal{T}$ is allowed to make mistakes by suspecting processes before they crash (see the scenario of Figure 4.1).

By contradiction, assume that there exists a reduction algorithm $T_{\mathcal{T}\rightarrow\mathcal{P}}$ that, for any failure pattern $F \in \mathcal{E}$ and any history $H_{\mathcal{T}} \in \mathcal{T}(F)$, constructs a history $H_{\mathcal{P}}$ such that $H_{\mathcal{P}} \in \mathcal{P}(F)$.

Consider failure pattern $F_1 \in \mathcal{E}$ such that $F_1(0) = \{j\}$, *correct*$(F_1) = \Pi - \{j\}$ (the only faulty process j is initially crashed) and take a history $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$ such that $H^1_{\mathcal{T}}(i,t) = \{j\}$, $\forall i \neq j, \forall t \in \mathbb{T}$. Consider run $R_1 = \langle F_1, H^1_{\mathcal{T}}, I, S_1, T\rangle$ of $T_{\mathcal{T}\rightarrow\mathcal{P}}$ that outputs a history $H^1_{\mathcal{P}} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists k_0 \in \mathbb{N}, \exists l \in \Pi - \{j\}$: $H^1_{\mathcal{P}}(l, T[k_0]) = \{j\}$.

Consider failure pattern $F_2 \in \mathcal{E}$ such that *correct*$(F_2) = \Pi$ (F_2 is failure-free) and define a history $H^2_{\mathcal{T}}$ such that $\forall i \in \Pi$ and $\forall t \in \mathbb{T}$:

$$H^2_{\mathcal{T}}(i,t) = \begin{cases} \{j\}, & t \leq T[k_0] \\ \emptyset, & t > T[k_0] \end{cases}$$

Note that $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$, and $\forall t \leq T[k_0]$, $\forall i \in \Pi - \{j\}$: $H^1_{\mathcal{T}}(i,t) = H^2_{\mathcal{T}}(i,t)$. Consider run $R_2 = \langle F_2, H^2_{\mathcal{T}}, I, S_2, T\rangle$ of $T_{\mathcal{T}\rightarrow\mathcal{P}}$ such that $S_1[k] = S_2[k], \forall k \leq k_0$ (processes take the same steps in R_1 and R_2 up to time $T[k_0]$). Let R_2 output a history $H^2_{\mathcal{P}} \in \mathcal{P}(F_2)$. Since partial runs of R_1 and R_2 for $t \leq T[k_0]$ are identical, the resulting history $H^2_{\mathcal{P}}$ is such that $H^2_{\mathcal{P}}(l, T[k_0]) = \{j\}$, for some $l \in \Pi - \{j\}$. But process j is alive at $T[k_0]$ in F_2, i.e., the strong accuracy property of $\mathcal{P}$ is violated — a contradiction.

Thus, $\mathcal{T} \prec_{\mathcal{E}} \mathcal{P}$. □

Proposition 4.2 *Let $\mathcal{E}$ be any environment that, for some process j, contains all failure patterns in which j crashes, and a failure-free failure pattern. Then $\Diamond\mathcal{P} \prec_{\mathcal{E}} \mathcal{T}$.*

Proof: $\Diamond\mathcal{P} \preceq_{\mathcal{E}} \mathcal{T}$ in any environment $\mathcal{E}$: by definition, every $\mathcal{T}$ satisfies strong completeness and eventual strong accuracy.

Now we show that $\mathcal{T}$ is not weaker than $\Diamond\mathcal{P}$. Intuitively, this follows from the fact that $\mathcal{T}$ is allowed to make only a *bounded* number of mistakes, while the number of mistakes $\Diamond\mathcal{P}$ can make is unbounded.

By contradiction, assume that there exists a reduction algorithm $T_{\Diamond\mathcal{P}\rightarrow\mathcal{T}}$ that, for any failure pattern $F \in \mathcal{E}$ and any history $H_{\Diamond\mathcal{P}} \in \Diamond\mathcal{P}(F)$, constructs a history $H_{\mathcal{T}}$ such that $H_{\mathcal{T}} \in \mathcal{T}(F)$.

Consider a failure-free pattern $F_1 \in \mathcal{E}$ (*correct*$(F_1) = \Pi$) and take $H^1_{\Diamond\mathcal{P}} \in \Diamond\mathcal{P}(F_1)$ such that $\forall i, \forall t \in \mathbb{T}$: $H^1_{\Diamond\mathcal{P}}(i,t) = \emptyset$. Consider a run $R_1 = \langle F_1, H^1_{\Diamond\mathcal{P}}, I, S_1, T\rangle$ of $T_{\Diamond\mathcal{P}\rightarrow\mathcal{T}}$ that outputs a history $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$. By the eventual strong accuracy property of $\mathcal{T}$, $\exists k_0 \in \mathbb{N}$, such that $\forall k \geq k_0$ and $\forall i \in \Pi$: $H^1_{\mathcal{T}}(i, T[k]) = \emptyset$.

Now consider a failure pattern $F_2 \in \mathcal{E}$ such that *correct*$(F_2) = \Pi - \{j\}$ and j crashes at time $T[k_0] + 1$. Take a history $H^2_{\Diamond\mathcal{P}} \in \Diamond\mathcal{P}(F_2)$ such that for all $t \in \mathbb{T}$ and $i \in \Pi$:

$$H^2_{\Diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\Diamond\mathcal{P}}(i,t), & t \leq T[k_0] \\ \{j\}, & t > T[k_0] \end{cases}$$

Now consider a run $R_2 = \langle F_2, H^2_{\diamond\mathcal{P}}, I, S_2, T\rangle$ of $T_{\diamond\mathcal{P}\to\mathcal{T}}$ that outputs a history $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$. Assume that $S_1[k] = S_2[k]$, $\forall k \leq k_0$. By definition, for all $i \in \Pi$, $H^2_{\mathcal{T}}(i, T[k_0]) = \emptyset$. By the strong completeness property of $\mathcal{T}$, there exists a time $k_1 > k_0$ such that $\forall i \neq j$: $H^2_{\mathcal{T}}(i, T[k_1]) = \{j\}$.

Now we construct a history $H^3_{\diamond\mathcal{P}}$ such that for all $t \in \mathbb{T}$ and $i \in \Pi$:

$$H^3_{\diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\diamond\mathcal{P}}(i,t), & t \leq T[k_0] \\ H^2_{\diamond\mathcal{P}}(i,t), & T[k_0] < t \leq T[k_1] \\ \emptyset, & t > T[k_1] \end{cases}$$

Clearly, $H^3_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_1)$.

Finally, consider a run $R_3 = \langle F_1, H^3_{\diamond\mathcal{P}}, I, S_3, T\rangle$ of $T_{\diamond\mathcal{P}\to\mathcal{T}}$ that outputs a history $H^3_{\mathcal{T}} \in \mathcal{T}(F_1)$. Assume that $S_3[k] = S_2[k]$, $\forall k \leq k_1$. Since partial runs of R_2 and R_3 for $t \leq T[k_1]$ are identical, there exists $i \neq j$ such that:

$$H^3_{\mathcal{T}}(i, T[k_0]) = \emptyset,$$
$$H^3_{\mathcal{T}}(i, T[k_1]) = \{j\}.$$

In other words, j is suspected by i at time $T[k_1]$ after being trusted by i at time $T[k_0] < T[k_1]$. By the trusting accuracy property of $\mathcal{T}$, j is crashed in F_1, which contradicts the assumption that F_1 is failure-free.

Thus, $\diamond\mathcal{P} \prec_{\mathcal{E}} \mathcal{T}$. □

4.3 The necessary condition for solving FTME

This section shows that the trusting failure detector $\mathcal{T}$ is necessary to solve FTME in *any* environment $\mathcal{E}$. In other words, we show that if a failure detector $\mathcal{D}$ solves FTME in $\mathcal{E}$, then $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{D}$.

Assume that an algorithm $\mathcal{A}$ solves FTME in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$. A reduction algorithm $T_{\mathcal{D}\to\mathcal{T}}$ that transforms $\mathcal{D}$ into $\mathcal{T}$ is presented in Figure 4.2. At any time $t \in \mathbb{T}$ and for any process $i \in \Pi$, $T_{\mathcal{D}\to\mathcal{T}}$ outputs the set of processes suspected by i, denoted $\mathcal{T}\text{-}output_i(t)$.

In the algorithm of Figure 4.2, processes can access n different critical sections. For convenience, we denote by CS_j the critical section of j-th instance of $\mathcal{A}$, where $j = 1, 2, \ldots, n$. Let try_{ij}, crit_{ij}, exit_{ij} and rem_{ij} denote, respectively, trying, critical, exit and remainder sections of process i with respect to CS_j, $j = 1, 2, \ldots, n$. By definition, if CS_j is used correctly (the processes are well-formed with respect to CS_j), then $\mathcal{A}$ guarantees the properties of FTME.

The idea of the algorithm is the following. Initially, $\forall i \in \Pi$: $\mathcal{T}\text{-}output_i = \Pi$ (every process is suspected). Process i first runs the trying protocol try_{ii} in order to enter CS_i. Since i is the only process in the trying section for CS_i, i eventually either crashes, or enters CS_i and then sends the message $[i, i]$ to all. Every process j that received $[i, i]$ stops suspecting i and executes try_{ji} in order to enter CS_i.

In our algorithm, a process can leave its CS only because of a crash. Thus, the only reason for which a process i can enter CS_j $(i \neq j)$ is the crash of j. In this case, process i sends the message $[i, j]$ to all processes. Every process that receives the message $[m, i, j]$ $(i \neq j)$ starts suspecting j.

As a result, eventually, no correct process is suspected by any correct process and every crashed process is permanently suspected by every correct process. Moreover, the only reason to start suspecting a process i after having trusted i, is a crash of i. That is, the output of $\mathcal{T}$ is extracted.

To ensure progress of the failure detector output, the reduction algorithm of Figure 4.2 maintains, at every process $i \in \Pi$, $n+2$ parallel tasks:

- **task** 0 in which i runs the trying protocol try_{ii};
- **task** k ($k = 1, 2, \ldots, n$) in which i detects that k has entered CS_k, stops suspecting k and runs the trying protocol try_{ik} (lines 9–10 are executed atomically);
- **task** $n+1$ in which i detects failures of other processes and starts suspecting them.

```
 1: $\mathcal{T}\text{-}output_i \leftarrow \Pi$                          { Initialization }
 2: $crashed_i \leftarrow \emptyset$
 3: start tasks $0, \ldots, n+1$

 4: task 0:
 5:    $\mathsf{try}_{ii}$                                        { $i$ requests $CS_i$ }
 6:    send $[i, i]$ to all                             { $i$ enters $CS_i$ }

                          {An indication that $k$ entered $CS_k$ is received }
 7: task $k$ ($k = 1, 2, \ldots, n$):
 8:    upon receive $[k, k]$ do
 9:       if $k \notin crashed_i$ then
10:          $\mathcal{T}\text{-}output_i \leftarrow \mathcal{T}\text{-}output_i - \{k\}$     { $i$ stops suspecting $k$ }
11:       if $k \neq i$ then
12:          $\mathsf{try}_{ik}$                                  { $i$ requests $CS_k$ }
13:          send $[i, k]$ to all                       { $i$ enters $CS_k$ }

                          {An indication that $j$ entered $CS_k$ is received }
14: task $n+1$:
15:    upon receive $[j, k]$ with $j \neq k$ do
16:       $crashed_i \leftarrow crashed_i \cup \{k\}$
17:       $\mathcal{T}\text{-}output_i \leftarrow \mathcal{T}\text{-}output_i \cup \{k\}$    { $i$ starts suspecting $k$ }
```

Figure 4.2: Reduction algorithm $T_{\mathcal{D}\rightarrow\mathcal{T}}$: code for each process i

Theorem 4.3 *For any environment $\mathcal{E}$, if a failure detector $\mathcal{D}$ solves FTME in $\mathcal{E}$, then $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{D}$.*

Proof: The algorithm of Figure 4.2 extracts the trusting failure detector $\mathcal{T}$ from any failure detector that solves FTME. According to the algorithm, no process i requests twice the same instance CS_j or exits. Thus, each i is well-formed with respect to each CS_j. Note that, once it has entered CS_j, i can leave CS_j only if i crashes.

By contradiction, assume that the strong completeness property of $\mathcal{T}$ is violated. More precisely,

$$\exists F,\ \exists i \in \mathit{correct}(F),\ \exists j \notin \mathit{correct}(F) : \forall t, \exists t' > t, j \notin \mathcal{T}\text{-}\mathit{output}_i(t').$$

By the algorithm, initially, $j \in \mathcal{T}\text{-}\mathit{output}_i$, and the correct process i removes j from $\mathcal{T}\text{-}\mathit{output}_i$ (line 10 of Figure 4.2) at most once and only if (a) the message $[j, j]$ is received (line 8), i.e., at some earlier time, l was in CS_j (line 6) and, (b) $j \notin crashed_i$.

If and when it happens, i runs try_{ij} in order to enter CS_j (line 12). By the *progress* property of FTME, at some time later, some correct process m is in CS_j. By the algorithm, m sends $[m, j]$ to all. Eventually, process i receives $[m, j]$ (j is faulty, thus, $m \neq j$). Since lines 9–10 are executed atomically, i cannot execute line 16 (while processing $[m, j]$) *before* executing line 10 (while processing $[j, j]$). As a result of processing $[m, j]$, i adds j to $\mathcal{T}\text{-}\mathit{output}_i$ (line 17) and, since j has already been inserted into $\mathit{crashed}_i$ and is never removed from there, j stays in $\mathcal{T}\text{-}\mathit{output}_i$ forever — a contradiction.

Thus, strong completeness of $\mathcal{T}$ is satisfied.

By contradiction, assume that the trusting accuracy of $\mathcal{T}$ is violated. More precisely,

$$\exists F,\ \exists i,\ \exists t, t',\ \exists j \notin F(t') : (t < t' \wedge j \notin \mathcal{T}\text{-}\mathit{output}_i(t) \wedge j \in \mathcal{T}\text{-}\mathit{output}_i(t')).$$

By the algorithm, i suspects j at time t' only if some process $k \neq j$ enters CS_j at some time $t_0 < t'$ and only if, at some time $t_1 < t_0$, j itself entered CS_j. By the *mutual exclusion* property of FTME, j had to leave CS_j *before* t_0. Since j never executes the exit protocol, j could leave CS_j only because of its crash, that is, $j \in F(t')$ — a contradiction.

By contradiction, assume now that eventual strong accuracy is violated. More precisely,

$$\exists F, \exists i \in \mathit{correct}(F), \exists j \in \mathit{correct}(F), \forall t, \exists t' > t : j \in \mathcal{T}\text{-}\mathit{output}_i(t').$$

Note that the assumption implies that $\forall t \in \mathbb{T}, j \in \mathcal{T}\text{-}\mathit{output}_i(t)$, otherwise, trusting accuracy is violated.

Thus, i never stops suspecting j: by the algorithm, i never reaches line 10 while processing the reception of $[j, j]$. That is, either (1) i receives $[k, j]$ with $k \neq j$ and put j into $\mathit{crashed}_i$ (lines 15-17), or (2) i never receives $[j, j]$.

Assume that (1) is true. By the algorithm, $[k, j]$ with $k \neq j$ can be only received if k entered CS_j at some time t_0 and if, at some time $t_1 < t_0$, j entered CS_j. Since j never executes the exit protocol, j could leave CS_j only if it is faulty — a contradiction.

Assume that (2) is true. Since both i and j are correct, j never sends $[j, j]$ (line 6). Thus, no process ever receives $[j, j]$. By the algorithm, a process k executes the trying protocol try_{kj} only if k received $[j, j]$. Thus, j is the only correct process that ever requests access to CS_j. By the *progress* property of FTME, j eventually enters CS_j and sends $[j, j]$ to all — a contradiction. Thus, the reduction algorithm of Figure 4.2 guarantees the properties of $\mathcal{T}$. □

Note that $T_{\mathcal{D}\rightarrow\mathcal{T}}$ does not use the full generality of FTME: in the algorithm, exit protocol is never used, and every process executes its trying protocol with respect to every CS_j at most once. This suggests that $\mathcal{T}$ can be extracted from a weaker problem than FTME which can be seen as a form of leader election.

4.4 The sufficient condition for solving FTME

We describe in Figure 4.3 an algorithm that solves FTME using $\mathcal{T}$ assuming any environment $\mathcal{E}$ in which a majority of processes are correct. The algorithm uses the fact that $\diamond\mathcal{P} \preceq_{\mathcal{E}} \mathcal{T}$ and, as a result, we can implement *total order broadcast* using $\mathcal{T}$ in $\mathcal{E}$ [15].

Total order broadcast is defined through the primitives to-broadcast() and to-deliver() and satisfies the following properties:

validity: if a correct process i to-broadcasts a message m, then i eventually to-delivers m;

agreement: if a process to-delivers a message m, every correct process eventually to-delivers m;

integrity: each process to-delivers every message at most once, and only if the message was previously to-broadcast;

total-order: if a process i to-delivers a message m before having to-delivered a message m', then no process j can to-deliver m' without having to-delivered m first.[1]

Note that the properties above imply that for any processes i, j and any times t, t', either the sequence of messages to-delivered by i at time t is a prefix of the sequence of messages to-delivered by j at time t', or vice-versa. In particular, if a process i to-delivered a message m at time t and a process j to-delivered a message m' at time t', then either m is to-delivered by j before t' or m' is to-delivered by i before t.

The algorithm of Figure 4.3 assumes that:

- an algorithm implementing total order broadcast is provided;
- every process i has access to the output of its trusting failure detector module $\mathcal{T}_i$;
- every process i is well-formed.

It is assumed that the underlying total order broadcast algorithm maintains at each process i a local variable *TO-queue*$_i$, initially empty FIFO queue. In a background task, whenever i to-delivers m, it atomically adds m to the end of *TO-queue*$_i$. In addition, each process i maintains the following local variables:

[1] This definition of the total-order property is slightly stronger than the one proposed in [39]: we require that all correct processes deliver the same sequence of messages, and all faulty processes deliver *prefixes* of this sequence. This distinction however does not matter for our results, since the algorithm given in [15] implements the strongest version of total order broadcast.

```
 1: $ready_i \leftarrow false$                                    { Initialization }
 2: $r_i \leftarrow 0$
 3: $trusted_i \leftarrow \emptyset$
 4: start tasks $0, \ldots, n$

    Trying protocol try$_i$:
 5: if not $ready_i$ then
 6:    send $[me, i]$ to all                            { Send a trust request to all }
 7:    wait until received $\lfloor n/2 \rfloor + 1$ $[ack]$'s
 8:    $ready_i \leftarrow true$
 9: $r_i \leftarrow r_i + 1$
10: to-broadcast($[i, r_i]$)
11: repeat
12:    wait until $TO\text{-}queue_i$ is not empty
13:    $[j, k] \leftarrow pop(TO\text{-}queue_i)$                { Get the next waiting request }
14:    if $i \neq j$ then
15:       wait until received $[exit, j, k]$ or received $[crash, j]$
16: until $i = j$
17: { $i$ enters CS }

    Exit protocol exit$_i$:
18: send $[exit, i, r_i]$ to all

                                                   { A crash of process $l$ is detected }
19: task 0:
20:    upon $(l \in trusted_i$ and $l \in \mathcal{T}_i)$ do
21:       $trusted_i \leftarrow trusted_i - \{l\}$
22:       send $[crash, l]$ to all                          { $l$ stops being trusted }

                                          { A trust request is received from $m \in \Pi$ }
23: task $m$ $(m = 1, 2, \ldots, n)$:
24:    upon receive $[me, m]$ do
25:       wait until $m \notin \mathcal{T}_i$                      { Wait until $m$ is trusted }
26:       $trusted_i \leftarrow trusted_i \cup \{m\}$
27:       send $[ack]$ to $m$
```

Figure 4.3: FTME algorithm using $\mathcal{T}$: code for each process i

1. a boolean $ready_i$, initially *false*, indicating whether i is ready to execute the trying protocol;

2. a set $trusted_i \subseteq \Pi$, initially empty, of processes currently trusted by i;

3. an integer r_i, initially 0, indicating the number of times i has run the trying protocol;

4. integers j and k indicating the last processed request of the type $[j, k]$ where j is the process that issued the request and k is j's request number.

The algorithm also assumes that every process i stores the identifiers of all received messages in a buffer, so that, for a given message m, the predicate "received m" (lines 7 and 15 of Figure 4.3) is true if and only if m has been previously received by i.

The idea of our algorithm is inspired by the well-known Bakery algorithm of Lamport [49, 51]: the processes that wish to enter their CSs (the candidates) first draw tickets and then are served in the order of their tickets' numbers. Before drawing a ticket, every candidate asks for a permission to proceed from some *correct* process and waits (line 7) until the permission is received (this eventually happens due to the assumption that a majority of processes are correct). Then the candidate is put into the waiting queue implemented by the total order broadcast mechanism. Total order broadcast guarantees that the requests are eventually delivered in the same order (line 12), i.e., no candidate i can be served unless every candidate in the waiting queue before i has been served and has released the resource, or crashed (line 15). If a process crashes in its CS, then at least one correct process will eventually detect the crash and inform the others (lines 20-22).

To ensure the *progress* property of FTME, in addition to the trying and exit protocols (respectively, lines 5–17 and line 18), the algorithm maintains, at every process $i \in \Pi$, $n+1$ parallel tasks:

- **task** 0 in which i detects failures of other processes;
- **task** m ($m = 1, 2, \dots, n$) in which i takes care of the trust request of process m.

We prove the correctness of the algorithm through Lemmas 4.4 and 4.5.

Lemma 4.4 *No two different processes are in their CSs at the same time.*

Proof: By contradiction, assume that i and j ($i \neq j$) are in their CSs at time t_0. Let, at time t_0, $r_i = k_i$ and $r_j = k_j$.

In the trying protocol (lines 5–17), every process to-broadcasts its request for a CS and no process enters its CS before having first to-delivered its request. Thus i must have to-delivered $[i, k_i]$ and j must have to-delivered $[j, k_j]$ before t_0. By the properties of total order broadcast, either j to-delivered $[i, k_i]$ before having to-delivered $[j, k_j]$, or the reverse. Assume, without loss of generality, that to-deliver($[i, k_i]$) precedes to-deliver($[j, k_j]$) at j. That is, at some time $t_1 < t_0$, j passed the "wait" clause in line 15 while processing $[i, k_i]$. Thus, one of the following events occurred *before* t_1 at j:

(1) j received $[exit, i, k_i]$: by the algorithm, i left the CS with $r_i = k_i$ before time t_1. But i is in the CS with $r_i = k_i$ at $t_0 > t_1$ — a contradiction.

(2) j received $[crash, i]$: by the algorithm, at some process m and time $t_2 < t_1$ the following is true: $i \in trusted_m$ and $i \in \mathcal{T}_m$. But i can be in $trusted_m$ only if previously $i \notin \mathcal{T}_m$ (lines 25–26). That is, m stopped trusting i at time t_2. By the trusting accuracy property of $\mathcal{T}$, i is crashed at t_2. But i is in the CS at $t_0 > t_2$ — a contradiction.

Hence, mutual exclusion is guaranteed. □

Lemma 4.5 *If a correct process is in its trying section, then at some later time some correct process is in its CS. If a correct process is in its exit section, then at some later time it enters its remainder section.*

Proof: Assume that a correct process $\bar{i}$ in its trying section at some time t_c with $r_{\bar{i}} = \bar{r}$, and no correct process is ever in its CS after t_c. By the algorithm, $\bar{i}$ never reaches line 17. Thus, $\bar{i}$ is blocked in a "wait" clause or at the non-terminating repeat-until loop. The first "wait" clause (line 7 of Figure 4.3) is not able to block the process, due to eventual strong accuracy of $\mathcal{T}$ and the fact that at least $\lfloor n/2 \rfloor + 1$ processes are correct. Thus, $\bar{i}$ eventually issues to-broadcast($[\bar{i}, \bar{r}]$). The second "wait" clause (more precisely, the statement in line 12 of Figure 4.3) is not blocking either, because of validity of total order broadcast: eventually, i to-delivers at least one message – $[\bar{i}, \bar{r}]$. Further, if the "wait" clause in line 15 is not blocking, then validity of total order broadcast implies that $[\bar{i}, \bar{r}]$ is eventually to-delivered by $\bar{i}$, thus $\bar{i}$ exits the repeat-until loop and enters its CS.

Thus, $\bar{i}$ is blocked in the third "wait" clause (line 15 of Figure 4.3) while processing some $[\bar{j}, \bar{k}]$ $(\bar{i} \neq \bar{j})$. Thus, $\bar{i}$ never receives $[\mathit{exit}, \bar{j}, \bar{k}]$ or $[\mathit{crash}, \bar{j}]$.

By integrity of total order broadcast, $\bar{j}$ has previously to-broadcast $[\bar{j}, \bar{k}]$ (line 10 of Figure 4.3).

Let j be any process that completes line 10.

We observe first that (**Claim 1**) j has been previously put in $\mathit{trusted}_m$ by some *correct* process m. Indeed, j received $\lfloor n/2 \rfloor + 1$ $[\mathit{ack}]$'s from processes that trusted j. Since at least $\lfloor n/2 \rfloor + 1$ processes are correct, j receives at least one $[\mathit{ack}]$ from a correct process m that previously put j in $\mathit{trusted}_m$ at some time t_0.

Then we notice that (**Claim 2**) if j is faulty, then every correct process eventually receives $[\mathit{crash}, j]$. Indeed, if j is faulty, then, by trusting completeness of $\mathcal{T}$, every correct process (and, in particular, the correct process m of Claim 1) eventually and permanently suspects j, i.e., $\exists t_1 > t_0 : \forall t > t_1 : j \in \mathcal{T}_m$. Thus, eventually, the condition of line 20 is satisfied at m for j ($j \in \mathit{trusted}_m$ and $j \in \mathcal{T}_m$). Thus, m sends $[\mathit{crash}, j]$ to all processes and every correct process eventually receives it.

Hence, process $\bar{j}$ is necessarily correct. Indeed, if $\bar{j}$ is faulty, then, by Claim 2, correct process $\bar{i}$ eventually receives $[\mathit{crash}, \bar{j}]$ and releases from waiting in line 15.

Further, we observe that trusting accuracy of $\mathcal{T}$ implies that (**Claim 3**) if a message $[\mathit{crash}, j]$ is received, then j is crashed.

Finally, we show that (**Claim 4**) if a correct process m passed an entry $[j, k]$ in the total order (is not blocked in line 15 while processing $[j, k]$), then no correct process can be blocked while processing $[j, k]$. Indeed, the following cases are possible:

(a) $j = m$: j enters its CS (line 17). By the assumption of the proof, no correct process is in its CS after t_c, thus, j left its CS before t_c and j sent $[\mathit{exit}, j, k]$ to all (line 18). Thus, every correct process eventually receives the message and releases.

(b) $j \neq m$, and j is faulty. By Claim 2, every correct process eventually receives $[\mathit{crash}, j]$ and releases.

(c) $j \neq m$, and j is correct. By Claim 3, m could only receive $[\mathit{exit}, j, k]$. Every correct process eventually receives $[exit, j, k]$ and releases.

Recall that $\bar{i}$ is blocked in line 15 while processing request $[\bar{j}, \bar{k}]$ ($\bar{i} \neq \bar{j}$). By Claim 2, $\bar{j}$ is correct, and, by Claim 4, $\bar{j}$ should have passed all entries in the total order that $\bar{i}$ has passed before reaching $[\bar{j}, \bar{k}]$. By the algorithm $\bar{j}$ enters its CS (line 17). By the assumption of the proof, no correct process is in its CS after t_c, thus, $\bar{j}$ left its CS before t_c and sent $[exit, \bar{j}, \bar{k}]$ to all. $\bar{i}$ eventually receives the message and releases — a contradiction.

The second part of the lemma follows directly from the algorithm: every correct process i that runs exit_i enters rem_i after a finite number of steps. That is, every correct process in its exit section eventually enters its remainder section.

Thus, progress is guaranteed. □

The following theorem follows directly from Lemmas 4.4 and 4.5:

Theorem 4.6 *The algorithm of Figure 4.3 solves FTME using $\mathcal{T}$, in any environment in which a majority of processes are correct.*

Finally, combining Theorem 4.3 and Theorem 4.6, we can state the following result:

Theorem 4.7 *For any environment $\mathcal{E}$ in which a majority of processes are correct, $\mathcal{T}$ is the weakest failure detector to solve FTME in $\mathcal{E}$.*

Remark. In fact, the algorithm of Figure 4.3 solves a seemingly harder problem: in addition to mutual exclusion and progress, it satisfies also the starvation-freedom property.

Indeed, assume that a correct process i is in its trying section with $r_i = k$. Eventually, due to the properties of the total order broadcast, all entities $[j, l]$ preceding $[i, k]$ in the total order are eventually processed: if any process in its CS eventually exits or crashes, no process can be blocked in a "wait" clause (see line 15 in Figure 4.3). Finally, i eventually reaches its own entry $[i, k]$ in the total order and i enters its CS.

From Theorem 4.7 it follows that any algorithm solving FTME in any environment $\mathcal{E}$ in which a majority of processes are correct can be transformed into an algorithm that solves FTME with the starvation freedom property in $\mathcal{E}$.

4.5 On the number of correct processes

By applying standard partitioning arguments, we can show that if it is possible to solve FTME using $\mathcal{T}$ in an environment $\mathcal{E}$, then the sets of correct processes of any two failure patterns in $\mathcal{E}$ intersect.

Proposition 4.8 *Let $\mathcal{E}$ be any failure pattern that includes a failure-free failure pattern, and two failure patterns F_1 and F_2 such that $correct(F_1) \cap correct(F_2) = \emptyset$. Then no algorithm solves FTME using $\mathcal{T}$ in $\mathcal{E}$.*

Proof: Assume that an algorithm $\mathcal{A}$ solves FTME using $\mathcal{T}$ in $\mathcal{E}$. Let $X = correct(F_1)$ and $Y = correct(F_2)$. Consider two possible runs of $\mathcal{A}$:

(1) R_1: no process in Y takes any step in R_1 (processes in Y are crashed in R_1), and processes in X always suspect every process in Y. Assume that a correct process $i \in X$ is the only process in its trying section. By the progress property of FTME, i enters its CS at some time t_1.

(2) R_2: no process from X takes any step in R_2 (processes in X are crashed in R_2), no process in Y takes any step before $t_1 + 1$, and processes in Y always suspect every process in X. Assume that a correct process $j \in Y$ is the only process in its trying section. By the *progress* property of FTME, j enters its CS at some time $t_2 > t_1$.

Assume that no process ever runs an exit protocol in R_1 or R_2. We construct a run R that is identical to R_1 at any time in $[0, t_1]$ and identical to R_2 at any time in $[t_1 + 1, t_2]$. Now assume that every process is correct in R, the processes in X and Y start to trust each other *after* t_2 (this is a valid history of $\mathcal{T}$), and all messages sent between X and Y are delayed until $t_2 + 1$. Evidently, R is a run of $\mathcal{A}$. But, since i and j never enter their exit sections, at time t_2 both i and j are in their CSs — a contradiction. □

Now let us consider *any* environment $\mathcal{E}$ in which a majority of correct processes is not necessarily guaranteed, and ask ourselves whether the perfect failure detector $\mathcal{P}$ is the weakest to solve FTME in $\mathcal{E}$. A close look at the correctness proof for the algorithm of Figure 4.3 reveals that we use the assumption of a correct majority only to implement the total order broadcast primitive and to guarantee that for each correct process i, there is a correct process m that trusts i ("wait" clause in line 7). If the *quorum* failure detector Σ [20] is available, we can overcome both issues even if up to $n - 1$ processes can crash. Indeed, total order broadcast is implementable in any environment using $(\mathcal{T}, \Sigma)$ [15, 20] and the "wait" clause in line 7 can be substituted by:

```
repeat
    $Q_i \leftarrow \Sigma_i$
until received [$ack$] from all $j \in Q_i$
```

Σ guarantees that eventually all processes in Σ_i are correct. On the other hand, by the eventual strong accuracy property of $\mathcal{T}$, every correct process is eventually trusted by all correct processes. Hence, this "wait" clause is non-blocking.

On the other hand, the properties of Σ imply that the output of Σ_i always includes at least one correct process. That is, if i received [ack] from every process in Σ_i, then at least one correct process m started trusting i. If i crashes while i is in its CS, m will eventually detect the crash (by stopping trusting i) and inform the other processes. Thus, we can implement FTME in any environment using failure detector $(\mathcal{T}, \Sigma)$. (For every failure pattern F, $(\mathcal{T}, \Sigma)$ outputs a pair of histories $(H_{\mathcal{T}}, H_{\mathcal{S}})$ ($\mathcal{R}_{(\mathcal{T},\Sigma)} = 2^{\Pi} \times 2^{\Pi}$), such that $H_{\mathcal{T}} \in \mathcal{T}(F)$ and $H_{\mathcal{S}} \in \mathcal{S}(F)$.)

We show that, in most environments, $\diamond\mathcal{S}$ is strictly weaker than $\mathcal{P}$. As for Propositions 4.1 and 4.2, the "weaker" part of the proof follows directly form the definition of $(\mathcal{T}, \Sigma)$, and the "strictly" part of the proof is done by contradiction: we assume that a reduction algorithm $T_{(\mathcal{T},\Sigma)\rightarrow\mathcal{P}}$ exists and expose a run of this reduction algorithm that violates some properties of $\mathcal{P}$.

Proposition 4.9 *Let $\mathcal{E}$ be any environment that contains a failure pattern in which some process initially crashes, and a failure-free failure pattern. Then* $(\mathcal{T}, \Sigma) \prec_{\mathcal{E}} \mathcal{P}$.

Proof:

(a) By Proposition 4.2 $\mathcal{T} \prec_{\mathcal{E}} \mathcal{P}$. It is straightforward to see that $\Sigma \preceq_{\mathcal{E}} \mathcal{P}$. That is, both $\mathcal{T}$ and Σ are weaker than $\mathcal{P}$. Thus, $(\mathcal{T}, \Sigma) \preceq_{\mathcal{E}} \mathcal{P}$.

(b) Now we show that $\mathcal{P}$ is not weaker than $(\mathcal{T}, \Sigma)$. Indeed, assume there exists an algorithm $T_{(\mathcal{T},\Sigma)\to\mathcal{P}}$ that, for any failure pattern $F \in \mathcal{E}$, constructs $H_{\mathcal{P}}$ from $H_{\mathcal{T}} \in \mathcal{T}(F)$ and $H_{\Sigma} \in \Sigma(F)$, such that $H_{\mathcal{P}} \in \mathcal{P}(F)$.

Let $j, l \in \Pi$ and $j \neq l$. Consider failure pattern $F_1 \in \mathcal{E}$ such that $F_1(0) = \{j\}$ (j is initially crashed), *correct*$(F_1) = \Pi - \{j\}$, and take histories $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$ and $H^1_{\Sigma} \in \Sigma(F_1)$ such that $\forall i \in \Pi, \forall t \in \mathbb{T}$: $H^1_{\mathcal{T}}(i,t) = \{j\}$ (j is always suspected) and $H^1_{\Sigma}(i,t) = \{l\}$. Assume that the corresponding run $R_1 = \langle F_1, (H^1_{\mathcal{T}}, H^1_{\Sigma}), I, S_1, T\rangle$ of $T_{(\mathcal{T},\Sigma)\to\mathcal{P}}$ outputs a history $H^1_{\mathcal{P}} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists k_0 \in \mathbb{N}$: $H^1_{\mathcal{P}}(l, T[k_0]) = \{j\}$.

Consider failure pattern $F_2 \in \mathcal{E}$ such that *correct*$(F_2) = \Pi$ and define histories $H^2_{\mathcal{T}}$ and H^2_{Σ} such that $\forall i \in \Pi$ and $\forall t \in \mathbb{T}$:

$$H^2_{\mathcal{T}}(i,t) = \begin{cases} \{j\}, & t \leq T[k_0] \\ \emptyset, & t > T[k_0] \end{cases}$$

$$H^2_{\Sigma}(i,t) = \begin{cases} \{l\}, & t \leq T[k_0] \\ \Pi, & t > T[k_0] \end{cases}$$

Clearly, $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$ and $H^2_{\Sigma} \in \Sigma(F_2)$.

Consider a run $R_2 = \langle F_2, (H^2_{\mathcal{T}}, H^2_{\Sigma}), I, S_2, T\rangle$ of $T_{(\mathcal{T},\Sigma)\to\mathcal{P}}$ that outputs a history $H^2_{\mathcal{P}} \in \mathcal{P}(F_2)$, where $S_1[k] = S_2[k], \forall k \leq k_0$. Thus, j takes no steps in S_2 for all $t \leq T[k_0]$. Since partial runs of R_1 and R_2 for $t \leq T[k_0]$ are identical, the resulting history $H^2_{\mathcal{P}}$ is such that $H^2_{\mathcal{P}}(l, T[k_0]) = \{j\}$. In other words, j is suspected before it crashes, and the strong accuracy property of $\mathcal{P}$ is violated.

By (a) and (b), we have $(\mathcal{T}, \Sigma) \prec_{\mathcal{E}} \mathcal{P}$. □

Hence, there is a failure detector $(\mathcal{T}, \Sigma)$ which is strictly weaker than $\mathcal{P}$ and is sufficient to solve FTME in any environment where up to $n-1$ processes can crash.

4.6 Group mutual exclusion

Group mutual exclusion [47, 48, 38] is a natural generalization of the classical mutual exclusion problem [23, 51], where a process requests a "session" before entering its critical section. Processes are allowed to be in their critical sections simultaneously provided that they have requested the same session. Sessions represent resources each of which can be accessed simultaneously by an arbitrary number of processes, but no two of which can be accessed simultaneously.

Formally, the trying protocol of process i has an integer parameter s. We say that *i requests session s at time t* if i is alive at t, and i is running the trying protocol $\mathsf{try}_i(s)$ or it is in its CS immediately after running $\mathsf{try}_i(s)$. As with FTME, we assume that every process i is well-formed.

Thus, in addition to the progress properties of FTME, fault-tolerant group mutual exclusion (FTGME) satisfies the group mutual exclusion and concurrent entering properties (we follow the terminology used in Section 4.1):

Progress:

(1) If a correct process is in its trying section, then at some time later some correct process is in its CS.

(2) If a correct process is in its exit section, then at some time later it enters its remainder section.

Group mutual exclusion: If two processes are in their critical sections at the same time, then they request the same session.

Concurrent entering: If a correct process i requests a session and no other process requests a different session, then i eventually enters its CS.

The last property means that, for a given session, a process that has *already* entered its CS cannot prevent another process requesting the same session from entering its CS. The property excludes trivial solutions of group mutual exclusion using any simple mutual exclusion algorithm. In contrast to [38, 70], we do not make the assumption that a process can stay in its CS for a finite time only. This is the reason why we put "eventually" instead of "a bounded number of its own steps" as in [38, 47, 70] in the concurrent entering property. Clearly, if another process is concurrently trying to enter a different session, it can enter its CS first. In this case, the trying process *can* prevent another process from entering its CS.

FTGME is at least as hard as FTME: we can easily implement FTME from FTGME by just associating every process with a unique session number. On the other hand, we show here that $\mathcal{T}$ solves FTGME in a system with a majority of correct processes. Thus, in the sense of failure detection, FTME and FTGME are equivalent.

In Figure 4.4, we present an algorithm that solves FTGME using $\mathcal{T}$. For each process i, the algorithm of Figure 4.4 defines trying protocol $\mathsf{try}_i(\mathit{session}_i)$ that handles the request of i for session $\mathit{session}_i$, and exit protocol exit_i. In the algorithm, each process i maintains the following local variables:

1. a boolean ready_i, initially *false*, indicating whether i is ready to execute the trying protocol;

2. an integer r_i, initially 0, indicating the number of requests for the CS that i has made;

3. a set $\mathit{trusted}_i$, initially empty, of processes currently trusted by i;

4. an integer ls_i, initially -1 (we assume that requested session numbers are non-negative), indicating the number of currently satisfied session;

5. a set inCS_i, initially empty, of requests with session number ls_i that i suspects to be currently satisfied;

6. integers j, k and s indicate the last processed request of type $[j, k, s]$ where j is the process that issued the request, k is j's request number and s is the session that j requests.

The algorithm is similar to that of Section 4.4. Before requesting a session every process waits until it gets trusted by a correct process. The requests are broadcast using the total order broadcast primitive to-broadcast(), and delivered through to-deliver(). It is assumed that a background task maintains at each process i a FIFO "waiting" queue $\mathit{TO\text{-}queue}_i$, initially empty. Whenever i to-delivers m, it atomically adds m to the end of $\mathit{TO\text{-}queue}_i$.

If several consecutive requests for the same session s are placed in the total order, then the requests are satisfied simultaneously. No request for a new session $s' \neq s$ is satisfied until all processes requested earlier session s leave their CSs.

Now we state the correctness of the algorithm through Lemmas 4.10–4.12.

Lemma 4.10 *If two processes are in their critical sections at the same time, then they request the same session.*

Proof: Assume that processes i and j requesting sessions s_i and s_j, respectively, are in their CSs at some time t_0. Let, at time t_0, $r_i = k_i$ and $r_j = k_j$.

In the trying protocol (lines 7–21), every process to-broadcasts its request for a CS and no process enters its CS before having first to-delivered its own request. Thus i must have to-delivered $[i, k_i, s_i]$ and j must have to-delivered $[j, k_j, s_j]$ before t_0. By the ordering property of to-broadcast, i and j can only to-deliver $[i, k_i, s_i]$ and $[j, k_j, s_j]$ in the same order. Assume, without loss of generality, That to-deliver($[i, k_i, s_i]$) precedes to-deliver($[j, k_j, s_j]$) at j.

By the algorithm, j can be in the CS with $\mathit{session}_j = s_j$ and $r_j = k_j$ at t_0 only if every entry $[j', k', s']$ with $s' \neq s_j$ in the total order preceding $[j, k_j, s_j]$ has passed through the "if" clause defined in lines 16–17 *before* time t_0. As a result, before time t_0, j has put (i, k_i) into inCS_j and set ls_j to s_i (lines 18 and 19).

Since i is still in its CS with $r_i = k_i$ at time t_0, j could not have received $[\mathit{exit}, i, k_i]$ before t_0. Now assume that j received $[\mathit{crash}, i]$ before t_0: by the algorithm of Figure 4.4, at some process m, at some time $t_1 < t_0$ the following is true: $i \in \mathit{trusted}_m$ and $i \notin \mathcal{T}_m$ (m stops trusting i). By trusting accuracy of $\mathcal{T}$, i is crashed at t_1. But i is in the CS at $t_0 > t_1$ — a contradiction.

Thus, j has not received $[\mathit{exit}, i, k_i]$ or $[\mathit{crash}, i]$ before t_0, i.e., the condition in line 27 is not satisfied at j before t_0. As a result, at the moment when j to-delivered $[j, k_j, s_j]$ (line 14), $(i, k_i) \in \mathit{inCS}_j$ and $\mathit{ls}_j = s_i$. Assume that j reaches line 16 while processing $[j, k_j, s_j]$ at some time $t_1 < t_0$ (j is in its CS at t_0). Furthermore, inCS_j is non-empty at any $t \in [t_1, t_0]$ (it includes at least one entry (i, k_i)), j never receives $[\mathit{crash}, j]$ (by trusting accuracy of $\mathcal{T}$), and $\mathit{ls}_j = s_i$ at t_1. Thus, j can pass lines 16–17 and enter its CS before t_0 only if $s_j = s_i$. Hence, the group mutual exclusion property of FTGME is guaranteed. □

Lemma 4.11 *If a correct process is in its trying section, then at some time later some correct process is in its CS. If a correct process is in its exit section, then at some time later it enters its remainder section.*

```
 1: $ready_i \leftarrow false$                                        { Initialization }
 2: $r_i \leftarrow 0$
 3: $trusted_i \leftarrow \emptyset$
 4: $inCS_i \leftarrow \emptyset$
 5: $ls_i \leftarrow -1$
 6: start tasks $0, \ldots, n$

    Trying protocol try$_i(session_i)$:
 7: if not $ready_i$ then
 8:   send $[me, i]$ to all                                  { Send a trust request }
 9:   wait until received $\lfloor n/2 \rfloor + 1$ $[ack]$'s
10:   $ready_i \leftarrow true$
11: $r_i \leftarrow r_i + 1$
12: to-broadcast($[i, r_i, session_i]$)
13: repeat
14:   wait until $TO\text{-}queue_i$ is not empty
15:   $[j, k, s] \leftarrow pop(TO\text{-}queue_i)$            { Get the next waiting request }
16:   if $inCS_i \neq \emptyset$ and $s \neq ls_i$ then
17:     wait until $inCS_i = \emptyset$ or received $[crash, j]$
18:   $inCS_i \leftarrow inCS_i \cup \{(j, k)\}$
19:   $ls_i \leftarrow s$
20: until $j = i$
21: { $i$ enters its CS }

    Exit protocol exit$_i$:
22: send $[exit, i, r_i]$ to all

23: task 0:
24:   upon ($j \in trusted_i$ and $j \in \mathcal{T}_i$) do
25:     $trusted_i \leftarrow trusted_i - \{j\}$            { A crash of process $j$ is detected }
26:     send $[crash, j]$ to all                           { $j$ stops being trusted }
27:   upon (($j, k) \in inCS_i$ and
                 (received $[exit, j, k]$ or received $[crash, j]$)) do
28:     $inCS_i \leftarrow inCS_i - \{(j, k)\}$             {$[j, k, ls_i]$ releases the CS }

29: task $m$ ($m = 1, 2, \ldots, n$):
30:   upon receive $[me, m]$ do
31:     wait until $m \notin \mathcal{T}_i$                 { A trust request is received from $m$ }
32:     $trusted_i \leftarrow trusted_i \cup \{m\}$          { $m$ is trusted by $i$ }
33:     send $[ack]$ to $m$
```

Figure 4.4: FTGME algorithm using $\mathcal{T}$: code for each process i

Proof: The proof is similar to the proof of Lemma 4.5. Assume that a correct process $\bar{i}$ is in its trying section at time t_0, and no correct process ever enters its CS after t_0. Applying the arguments of Lemma 4.5, we observe that $\bar{i}$ is blocked in line 17 of Figure 4.4 because some entry $(\bar{j}, \bar{k})$ never leaves $inCS_{\bar{i}}$ (line 28). Claims 1–4 of Lemma 4.5 are proved similarly. By Claim 1 and Claim 2 of Lemma 4.5, $\bar{j}$ must be correct. By Claim 3 and Claim 4 of Lemma 4.5 $\bar{j}$ should have passed all entries in the total order that precede $[\bar{j}, \bar{k}, \bar{s}]$ and entered its CS. Since no process is in its CS after t_0, $\bar{j}$ executed the exit protocol before t_0 and sent $[exit, \bar{j}, \bar{k}]$ to all. Thus $\bar{i}$ eventually receives $[exit, \bar{j}, \bar{k}]$ and releases — a contradiction. Hence, the progress property of FTGME is ensured. □

Lemma 4.12 *If a correct process i requests a session and no other process requests a different session, then i eventually enters its CS.*

Proof: Assume that, at time t_0, a process i requests a session s_i with $r_i = k_i$ and no other process requests a different session. Thus, all processes requesting different sessions have left their CSs or crashed before t_0. As a result, after some time, either $inCS_i = \emptyset$ or $ls_i = s_i$. By the algorithm, eventually, i starts processing its own request $[i, k_i, s_i]$ with $ls_i = s_i$ (lines 14–16) and enters its CS (line 21). Hence, the concurrent entering property of FTGME is ensured. □

Finally, using the results of Section 4.3, we can state the following theorem:

Theorem 4.13 *For any environment $\mathcal{E}$ in which a majority of processes are correct, $\mathcal{T}$ is the weakest failure detector to solve FTGME in $\mathcal{E}$.*

Remark. Similar to the FTME algorithm of Figure 4.3, our FTGME algorithm satisfies also the starvation freedom property.

Analogously, when $\lceil \frac{n}{2} \rceil$ or more processes can crash, we can solve FTGME with $(\mathcal{T}, \Sigma)$, simply by implementing total order broadcast using $(\mathcal{T}, \Sigma)$, and substituting line 9 of the algorithm in Figure 4.4 with:

```
repeat
    $Q_i \leftarrow \Sigma_i$
until received $[ack]$ from all $j \in Q_i$
```

4.7 Cost of resilience

In this section we compare the performance of our algorithm (Figure 4.3) with the well-known algorithms of [58] and [63]. (The algorithms of [58] and [63] were designed for the failure-free asynchronous model but could be ported into the crash-prone model assuming $\mathcal{P}$. More details on the comparative analysis of the algorithms of [58] and [63] are available in [67].)

The performance of mutual exclusion algorithms can be measured through the following metrics [67]: (a) the *bootstrapping delay*, which is the time required before a process runs its trying protocol for the first time; (b) the *number of messages* sent during the trying protocol, (c) the *synchronization delay*, which is the time required after a process leaves the CS and before the next process enters the CS, and (d) the *response time*, which is the time required for a process to complete the trying protocol. We also consider two special loading conditions: *low load* and *high load.* In low load conditions, there is seldom more than one request to enter the CS at a time in the system. In high load conditions, any process that leaves the CS immediately executes the trying protocol again. In discussing performance, we concentrate here on the runs where no process crashes (the most frequent runs in practice), which are usually called *nice* runs.

We denote by t_c the maximum message propagation delay, and e_c the maximum CS execution time. The bootstrapping delay of our algorithm (Figure 4.3) is bounded by $2t_c$: before processing any request for CS, every process should receive the acknowledgment from a majority of the processes. The algorithm has a relatively high message complexity: each request for CS requires $O(n^2)$ messages per CS invocation. The synchronization delay is bounded by t_c: that is, it requires only one communication step to inform the next waiting process that it can enter the CS. The response time in low load conditions is defined by the time to deliver a total order broadcast message – $2t_c$. At high loads, on the average, all other processes execute their CSs between two successive executions of the CS: the response time converges to $n(t_c + e_c)$.

The results of our comparative analysis are presented in Figure 4.5. The performance degradation due to the use of $\mathcal{T}$ reflects the longer bootstrapping delay which is inherent to the use of $\mathcal{T}$ and higher message complexity inherited from using total order broadcast. It would be interesting to figure out to what extent our algorithm of Figure 4.3 could be optimized, e.g., by breaking the encapsulation of the total order broadcast box.

Metrics	**Maekawa** [58]	**RA** [63]	**$\mathcal{T}$-based**
Bootstrapping delay	0	0	$2t_c$
Number of messages	Low	Moderate	High
Sync. delay	$2t_c$ (deadlock-prone) t_c (deadlock-free)	t_c	t_c
Response time			
low load	$2t_c$	$2t_c$	$2t_c$
high load	$n(2t_c + e_c)$	$n(t_c + e_c)$	$n(t_c + e_c)$

Figure 4.5: Comparative performance analysis of mutual exclusion algorithms

4.8 Implementing $\mathcal{T}$

Is it more beneficial in practice to use a mutual exclusion algorithm based on $\mathcal{T}$, instead of a traditional algorithm assuming $\mathcal{P}$? The answer is “yes, to some extent”. Indeed, if we translate the very fact of not trusting a correct process into a *mistake*, then $\mathcal{T}$ clearly tolerates mistakes whereas $\mathcal{P}$ does not. More precisely, $\mathcal{T}$ is allowed to make up to n^2 mistakes (up to n mistakes for each module $\mathcal{T}_i$, $i \in \Pi$). As a result, given synchrony assumptions, it is somewhat easier to implement $\mathcal{T}$ than $\mathcal{P}$.

For example, in a possible implementation of $\mathcal{T}$, every process i can, starting from 0, gradually increase the timeout t_{ij} corresponding to a heart-beat message sent to a process j until a response from j is received. Thus, every such t_{ij} can be flexibly adapted to the current network conditions. (Clearly, as soon as $\mathcal{T}$ starts trusting a process j, it is not allowed to make mistakes about j any more.)

In contrast, $\mathcal{P}$ does not allow this kind of “fine-tuning” of the timeouts: the timeouts are supposed to be known in advance. In order to minimize the probability of mistakes, the timeouts are normally chosen sufficiently large, and the choice is based on some a priori assumptions about current network conditions. This might exclude some remote sites from the group and violate the accuracy properties of the failure detector.

Thus, we can implement $\mathcal{T}$ in a more effective manner than $\mathcal{P}$, and an algorithm that solves FTME using $\mathcal{T}$ exhibits a smaller probability of violating the requirements of the problem, than one using $\mathcal{P}$, i.e., the use of $\mathcal{T}$ provides more resilience. As we have shown in Section 4.7, the performance cost of this resilience reflects the *bootstrapping delay*, i.e., the time a new process needs to enter its CS for the first time, and higher message complexity inherited from using total order broadcast.

4.9 Open questions

The question of the weakest failure detector for solving FTME in *all* environments remains open.

In Section 4.5, we showed that $(\mathcal{T}, \Sigma)$, a composition of the trusting failure detector $\mathcal{T}$ and the quorum failure detector Σ [20], is *sufficient* to solve FTME in all environments. However, by careful inspection of the algorithm in Figure 4.3, we observe that the strong communication guarantees provided by Σ do not seem to be necessary for solving FTME. Indeed, let us define a weaker form of Σ, denoted Σ^{me} ("me" stands for mutual exclusion), that outputs a set of processes at each process, called *me-quorum*. Eventually, every set consists of only non-faulty processes and if two me-quorums output at times t and t' $(t > t')$ at, respectively, p and q do not intersect, then q has crashed by time t. In other words, any me-quorum obtained by process p at time t is guaranteed to intersect with any me-quorum obtained earlier by any process which is *alive* at t. By employing the technique of Section 3.2, we believe we can show that Σ^{me} is *necessary* to solve FTME in any environment.

Consider a $(\mathcal{T}, \Sigma)$-based FTME algorithm (Section 4.5). If we substitute Σ with Σ^{me}, then we have a problem with the progress property of FTME. Indeed, now a process j can enter its CS without getting trusted by a correct process first. As a result, if j fails while in its CS, then there is no guarantee that some correct process will eventually detect the failure. However, if any process i observes that no process in Σ_i^{me} trusts j, and j previously "drew a ticket" by getting trusted by all processes in Σ_j^{me}, then it immediately follows that j has previously crashed. This can be used as a criterion to stop waiting and proceed to the next entry in the total order (line 15 in Figure 4.3).

Similarly, we conjecture that $(\mathcal{T}, \Sigma^{me})$ can be used to implement an "oblivious" form of total order broadcast, in which a process may "skip" delivering messages of already failed processes. This meets the specification of FTME: a process does not have to worry whether an already failed process could have previously entered its CS or not.

We thus conjecture that $(\mathcal{T}, \Sigma^{me})$ is the weakest failure detector to solve FTME in all environments.

4.10 Related work

The mutual exclusion problem has been extensively studied for the last few decades [23, 51, 62, 1, 59, 55, 67]. Traditionally, mutual exclusion algorithms either assume that no process crashes outside its remainder section [23, 51, 62, 55, 67], or suppose that (1) every crash is eventually detected by every correct process and (2) no correct process is suspected [1, 59]: the conjunction of (1) and (2) is equivalent to the assumption of the *perfect* failure detector $\mathcal{P}$. In other words, $\mathcal{P}$ has been shown *sufficient* to solve mutual exclusion in a fault-tolerant manner. However, to our knowledge, the question of whether $\mathcal{P}$ is also *necessary* to solve FTME remained open until now.

Chapter 5

Quittable Consensus and NBAC

In this chapter, the weakest failure detectors for solving quittable consensus (QC) and non-blocking atomic commit (NBAC) in all environments are determined.

Section 5.1 recalls the specification of QC [40]. Section 5.2 determines the weakest failure detector to solve QC. Section 5.3 recalls the specification of NBAC, establishes a close relationship between QC and NBAC, and uses this to determine the weakest failure detector to solve NBAC. Section 5.4 contains some concluding remarks. Section 5.5 discusses the related work.

The results of this chapter appeared originally in [21].

5.1 Quittable consensus (QC)

In the *quittable consensus* problem, each process p starts with an initial value $v \in V$ (we say that p *proposes* v), and terminates with a *decision* value in $V \cup \{\mathcal{Q}\}$ ($\mathcal{Q}$ stands for "quit") . It is required that:

Termination: If every correct process proposes a value, then every correct process eventually returns a decision.

Agreement: No two processes (whether correct or faulty) return different values.

Validity: A process may only decide a value $v \in V \cup \{\mathcal{Q}\}$. Moreover,
(a) If $v \in V$, then some process previously proposed v.
(b) If $v = \mathcal{Q}$, then a failure previously occurred.

Here V is an arbitrary set of at least two values. In the binary version of QC, $V = \{0, 1\}$, i.e., processes can propose values in $\{0, 1\}$ and decide values in $\{0, 1, \mathcal{Q}\}$.

Lemma 5.1 *Any solution to binary QC can be transformed into a solution to multivalued QC for any set of proposal values V.*

```
Procedure QCPROPOSE($v$): { $v$ can be any value }
send $[v, p]$ to all
$k \leftarrow 0$
repeat
    $c \leftarrow (k \mod n) + 1$
    if received $[v', p_c]$ then
        send $[v', p_c]$ to all
        $d \leftarrow$ BQCPROPOSE$_k(1)$ { run binary QC proposing 1 }
    else
        $d \leftarrow$ BQCPROPOSE$_k(0)$ { run binary QC proposing 0 }
    $k \leftarrow k + 1$
until $d \in \{1, \mathcal{Q}\}$
if $d = \mathcal{Q}$ return $\mathcal{Q}$
wait until $[v', p_c]$ is received
return $v'$
```

Figure 5.1: Transforming binary QC into multivalued QC: code for each process p

Proof: A simple algorithm that transforms any solution to the binary QC into a solution to a multivalued one is described in Figure 5.1. The algorithm employs the technique of [60]. Each process p sends its proposal $v \in V$ to all and then runs through a sequence of instances of the given binary QC algorithm. For each $k \in \mathbb{N}$, BQCPROPOSE$_k()$ denotes p's invocation of the k-th instance of the binary QC algorithm.

In each round k, if p previously received the proposal of process $p_{(k \mod n)+1}$, then p forwards the proposal to all and proposes 1, otherwise p proposes 0. If $\mathcal{Q}$ is returned in the k-th instance of the given binary QC algorithm, then p returns $\mathcal{Q}$. If 1 is returned, then p waits until the proposal of process $p_{(k \mod n)+1}$ is received and decides the proposal. Finally, if 0 is returned, then p proceeds to round $k + 1$.

The Validity property of QC follows immediately from the algorithm. The Agreement property of binary QC implies that the first instance of binary QC which returns a value in $\{1, \mathcal{Q}\}$ is the same at all processes. The Agreement property of QC follows. Since proposals sent by correct processes are eventually received by all correct processes, 1 or $\mathcal{Q}$ is eventually decided in some instance of binary QC algorithm. Since, before proposing 1 in instance k, process p first forwards the proposal of $p_{(k \mod n)+1}$ to all, the proposal is eventually received by all correct processes: the Termination property of QC follows. □

5.2 The weakest failure detector to solve QC

We define a new failure detector denoted Ψ and show that it is the weakest failure detector to solve QC in *any* environment. To prove this, we first show that Ψ can be used to solve QC in any environment. We then prove that, for every environment $\mathcal{E}$, any failure detector that can be used to solve QC in $\mathcal{E}$ can be transformed into Ψ in $\mathcal{E}$.

```
Procedure qcPropose($v$): { $v$ is 1 or 0 }
while $\Psi_p = \bot$ do nop
if $\Psi_p \in \{\mathbf{green}, \mathbf{red}\}$
  then { henceforth $\Psi$ behaves like $\mathcal{FS}$ }
    return $\mathcal{Q}$
  else { henceforth $\Psi$ behaves like $(\Omega, \Sigma)$ }
    $d \leftarrow$ consPropose($v$) { use $\Psi$ to run $(\Omega, \Sigma)$-based consensus algorithm }
    return $d$
```

Figure 5.2: Using Ψ to solve QC

5.2.1 Specification of failure detector Ψ

Roughly speaking, Ψ behaves as follows: For an initial period of time the output of Ψ at each process is $\bot$. Eventually, however, Ψ behaves either like the failure detector (Ω, Σ) at all processes, or, *in case a failure previously occurred*, it *may* instead behave like the failure detector $\mathcal{FS}$ at all processes. The switch from $\bot$ to (Ω, Σ) or $\mathcal{FS}$ need not occur simultaneously at all processes, but the same choice is made by all processes. Note that the switch from $\bot$ to $\mathcal{FS}$ is allowable *only* if a failure previously occurred. Furthermore, if a failure does occur processes are not *required* to switch from $\bot$ to $\mathcal{FS}$; they may still switch to (Ω, Σ).

More precisely, Ψ is defined as follows. For each failure pattern F,

$$\begin{aligned}
H \in \Psi(F) \quad \Leftrightarrow \quad & \Big(\exists H' \in (\Omega,\Sigma)(F)\ \forall p \in \Pi\ \exists t \in \mathbb{T} \\
& \big(\forall t' < t\ H(p,t') = \bot \wedge \forall t' \geq t\ H(p,t') = H'(p,t')\big)\Big) \vee \\
& \Big(\exists t \in \mathbb{T}\ \Big(F(t) \neq \emptyset\ \wedge\ \exists H' \in \mathcal{FS}(F)\ \forall p \in \Pi \\
& \big(\forall t' < t\ H(p,t') = \bot \wedge \forall t' \geq t\ H(p,t') = H'(p,t')\big)\Big)\Big)
\end{aligned}$$

5.2.2 Using Ψ to solve QC

It is easy to use Ψ to solve QC in any environment $\mathcal{E}$ (see Figure 5.2). Each process p waits until the output of Ψ becomes different from $\bot$. At that time, either Ψ starts behaving like $\mathcal{FS}$ or it starts behaving like (Ω, Σ). If Ψ starts behaving like $\mathcal{FS}$ (Ψ can do so *only* if a failure previously occurred), p returns $\mathcal{Q}$. The remaining case is that Ψ starts behaving like (Ω, Σ). It is shown in [20] that there is an algorithm that uses (Ω, Σ) to solve consensus in any environment. Therefore, in this case, processes propose their initial value to that consensus algorithm and return the value decided by that algorithm. In Figure 5.2, consPropose() denotes p's invocation of the algorithm that solves consensus using (Ω, Σ). Hence, we obtain the following result:

Theorem 5.2 *For all environments $\mathcal{E}$, Ψ can be used to solve QC in $\mathcal{E}$.*

```
Initially:
$\Psi$-$output_p \leftarrow \bot$ { $\Psi$-$output_p$ is the output of $p$'s module of $\Psi$ }
task 1:
do forever { This is done exactly as in [14] (Section 3.1) }
cobegin
    $p$ builds an ever-increasing DAG $G_p$ of failure detectors samples by repeatedly
        sampling its failure detector and exchanging samples with other processes.
||
    $p$ uses $G_p$ and the $n + 1$ initial configurations to construct a forest $\Upsilon_p$ of ever-increasing
        simulated runs of algorithm $\mathcal{A}$ using $\mathcal{D}$ that could have occurred
        with the current failure pattern $F$ and the current failure detector history $H \in \mathcal{D}(F)$.
coend
task 2:
wait until $p$ decides in some run of every tree of the forest $\Upsilon_p$
if $p$ decides $\mathcal{Q}$ in some run
    then
        $p$ executes $\mathcal{A}$ by proposing 0
    else { every tree of $\Upsilon_p$ has a run where $p$ decides 0 or 1 }
        let $I$ and $I'$ be initial configurations that differ only in the proposal of one process and
            $S$ and $S'$ be schedules in $\Upsilon_p$ so that $p$ decides 0 in $S(I)$ and 1 in $S'(I')$
        $p$ executes $\mathcal{A}$ by proposing $(I, I', S, S')$
wait until $p$ decides in this execution of $\mathcal{A}$
if $p$ decides 0 or $\mathcal{Q}$
    then { extract $\mathcal{FS}$ }
        $\Psi$-$output_p \leftarrow$ red
    else { $p$'s decision is of the form $(I_0, I_1, S_0, S_1)$ }
        $\Omega$-$output_p \leftarrow p$ ; $\Sigma$-$output_p \leftarrow \Pi$ { extract $(\Omega, \Sigma)$ }
        cobegin
            { extract $\Omega$ }
            do forever $\Omega$-$output_p \leftarrow$ id of the process that $p$
            extracts using $\Upsilon_p$ and the procedure described in [14]
        ||
            { extract $\Sigma$ }
            let $(I_0, I_1, S_0, S_1)$ be the decision value of $p$
            let $\mathcal{C}$ be the set of configurations reached by applying
                all prefixes of $S_0, S_1$ to $I_0, I_1$, respectively
            do forever
                wait until $p$ adds a new failure detector sample
                        $u$ to its DAG $G_p$
                repeat
                    let $G_p(u)$ be the subgraph induced by
                        the descendants of $u$ in $G_p$
                    for each $C \in \mathcal{C}$ construct the set $\mathcal{S}_C$ of all schedules compatible with
                        some path of $G_p(u)$ and applicable to $C$
                until for each $C \in \mathcal{C}$ there is a schedule $S \in \mathcal{S}_C$
                    such that $p$ decides in $S(C)$
                $\Sigma$-$output_p \leftarrow \bigcup_{C \in \mathcal{C}}$ set of processes that take steps
                    in the schedule $S \in \mathcal{S}_C$ such that $p$ decides in $S(C)$
        ||
            { combine $\Omega$ and $\Sigma$ to $\Psi$ }
            do forever $\Psi$-$output_p \leftarrow (\Omega$-$output_p, \Sigma$-$output_p)$
        coend
```

Figure 5.3: Extracting Ψ from $\mathcal{D}$ and QC algorithm $\mathcal{A}$: code for each process p

5.2.3 Extracting Ψ from any failure detector that solves QC

Let $\mathcal{D}$ be an arbitrary failure detector that can be used to solve QC in some environment $\mathcal{E}$. Let $\mathcal{A}$ be an algorithm that uses $\mathcal{D}$ to solve QC in environment $\mathcal{E}$. We must prove that Ψ can be "extracted" from $\mathcal{D}$ in environment $\mathcal{E}$, i.e., processes can run in $\mathcal{E}$ a transformation algorithm that uses $\mathcal{D}$ and $\mathcal{A}$ to generate the output of Ψ — a failure detector that initially outputs $\perp$ and later behaves either like (Ω, Σ) or like $\mathcal{FS}$. The reduction algorithm $T_{\mathcal{D}\rightarrow\Psi}$ is shown in Figure 5.3 and is explained below.

Overview of the reduction algorithm

Each process p starts by outputting $\perp$ (line 1). While doing so, p determines whether in the current run it is possible to extract (Ω, Σ), or it is legitimate to start behaving like $\mathcal{FS}$ and output **red** because a failure occurred.

In task 1, p simulates runs of $\mathcal{A}$ that *could have occurred* in the current failure detector history of $\mathcal{D}$ and the current failure pattern F, exactly as in [14] (see Section 3.1). It does this by "sampling" its local module of $\mathcal{D}$ and exchanging failure detector samples with the other processes (line 4). Process p organizes these samples into an ever-increasing DAG G_p whose edges are consistent with the order in which the failure detector samples were actually taken. Using G_p, p simulates ever-increasing partial runs of algorithm $\mathcal{A}$ that are compatible with paths in G_p (line 6). Each process p organizes these runs into a forest of $n+1$ trees, denoted Υ_p. For any i, $0 \leq i \leq n$, the i-th tree of this forest, denoted Υ_p^i, corresponds to simulated runs of $\mathcal{A}$ in which processes $p_1, \ldots, p_i$ propose 1, and $p_{i+1}, \ldots, p_n$ propose 0. A path from the root of a tree to a node S in this tree corresponds to the schedule of a partial run of $\mathcal{A}$, where every edge along the path corresponds to a step of some process.

In task 2, p waits until it decides in some simulated run of every tree of the forest Υ_p (line 8). If p decides $\mathcal{Q}$ in any of these runs, then a failure must have occurred (in the current failure pattern), and so p knows that it is legitimate to behave like $\mathcal{FS}$ by outputting **red** in this run. Otherwise (p's decisions in the simulated runs are 0s or 1s), p determines that it is possible to extract (Ω, Σ) in the current run.

At this point, p executes the given QC algorithm $\mathcal{A}$ (using failure detector $\mathcal{D}$) in order to *agree* with all the other processes on whether to output **red** or to extract (Ω, Σ). Specifically, if p has determined that it is legitimate to output **red** then it proposes 0 to $\mathcal{A}$ (line 11), otherwise it proposes (I, I', S, S') (line 14) where I and I' are initial configurations that differ only in the proposal of one process, and S and S' are schedules in Υ_p such that p decides 0 in $S(I)$ and 1 in $S'(I')$. The following lemma proves that these configurations and schedules exist.

Lemma 5.3 *If any process p reaches line 12 then there are initial configurations I and I', and schedules S and S' in Υ_p, such that (a) I and I' differ only in the proposal of one process, and (b) p decides* 0 *in $S(I)$ and* 1 *in $S'(I')$.*

Proof: If any process p reaches line 12, then in each tree of Υ_p, p has a run in which it decides 0 or 1. In the tree Υ_p^0 where every process proposes 0, p's decision must be 0. Respectively, in Υ_p^n, p's decision must be 1. The result immediately follows. □

If $\mathcal{A}$ returns 0 or $\mathcal{Q}$, then p stops outputting $\perp$ and outputs **red** from that time on (line 18). If $\mathcal{A}$ returns a value of the form (I_0, I_1, S_0, S_1), then p stops outputting $\perp$ and starts extracting Ω (line 22) and Σ (lines 24-32). Ω is extracted as in [14] (see Section "Extracting Ω" below). Σ is extracted using novel techniques explained in Section "Extracting Σ" below.

Note that processes use the given QC algorithm $\mathcal{A}$ and failure detector $\mathcal{D}$ in two different ways and for different purposes. First each process *simulates* many runs of $\mathcal{A}$ to determine whether it is legitimate to output **red** or it is possible to extract (Ω, Σ) in the current run. Then processes actually execute $\mathcal{A}$ (this is a *real* execution, not a simulated one) to reach a common decision on whether to output **red** or to extract (Ω, Σ). Finally, if processes decide to extract (Ω, Σ), they continue the simulation of runs of $\mathcal{A}$ to do this extraction.

Extracting Ω

To extract Ω, p must continuously output the id of a process such that, after some time, correct processes output the id of the same correct process. This is done using the procedure of [14] (see Section 3.1), with some minor differences explained below.

As in [14], because of the way each process p constructs its ever-increasing forest Υ_p of simulated runs, the forests of correct processes tend to the same infinite limit forest, denoted Υ. The limit tree of Υ_p^i is denoted Υ^i. Each node S of the limit forest Υ is tagged by the set of decisions reached by correct processes in partial runs that correspond to descendants of S.

In [14] the only possible decisions were 0 or 1, and so these were the only possible tags. Consequently, each node was 0-*valent*, 1-*valent* or *bivalent* (with two tags). Here there are three possible decisions (0, 1 or $\mathcal{Q}$) so each node is 0-*valent*, 1-*valent*, $\mathcal{Q}$-*valent* or *multivalent* (with two or three tags).

In [14] (and here) the extraction of the id of a common correct process relies on the existence of a *critical index* i in the limit forest Υ. Here we define i to be critical if the root of Υ^i is multivalent (in which case it is called *multivalent critical*), or if the root of Υ^{i-1} is u-valent and the root of Υ^i is v-valent, where $u, v \in \{0, 1, \mathcal{Q}\}$ and $u \neq v$ (in which case it is called *univalent critical*).

In [14] it is shown that a critical index always exists. In our case, however, this is not necessarily the case. If some process crashes (in the current failure pattern), it is possible that in all simulated runs of QC algorithm $\mathcal{A}$ in Υ all decisions are $\mathcal{Q}$. In this case, the roots of all trees in the limit forest Υ are tagged only with $\mathcal{Q}$. So there is no critical index, and we cannot apply the techniques of [14] to extract the id of a correct process! This is why, in our transformation algorithm, processes do not always attempt to extract Ω from $\mathcal{D}$. However, if a process actually attempts to extract Ω (in line 22) then a critical index does exist in the limit forest Υ, and so Ω can indeed be extracted:

Lemma 5.4 *If any process reaches line 22 then the limit forest Υ has a critical index.*

Proof: If a process reaches line 22, then it previously decided a tuple of the form (I_0, I_1, S_0, S_1) in line 15. Thus, by the Validity property of QC, some process q (not necessarily correct) proposed some tuple in line 14. We first show that the limit forest Υ has some run where some *correct* process decides a value other than $\mathcal{Q}$.

Since q proposed a tuple in line 14, it must have decided some value $v \neq \mathcal{Q}$ in some partial run R of $\mathcal{A}$ in Υ_q. Before q proposed its tuple in line 14, it sent to all processes the finite path of q's DAG (of failure detector samples) that gave rise to the partial run R. Thus, after receiving this path and integrating it in its own DAG, every correct process p also constructs partial run R (which is compatible with this path), and includes it in its own forest Υ_p. So the partial run R where q decides v is also embedded in the limit forest Υ. Note that Υ includes an infinite run R^* that extends R such that all the correct processes take an infinite number of steps in R^*. By the Termination and Agreement properties of $\mathcal{A}$, all the correct processes decide v (the same as q) in run R^*. So Υ has a run where all correct processes decide $v \neq \mathcal{Q}$.

From the above, the root of some tree Υ^j of the limit forest Υ has tag $v \neq \mathcal{Q}$. Without loss of generality, assume $v = 0$. Note that the root of tree Υ^n, where all processes propose 1, must have a tag $u \neq 0$ (it can be 1 or $\mathcal{Q}$). Therefore, some index i between j and n must be critical. □

Extracting Σ

To extract Σ, p must continuously output a set of processes (a quorum) such that the quorums of all processes always intersect, and eventually they contain only correct processes. This is done in lines 24-32 as follows.

When process p reaches line 24, it has agreed with other processes on two initial configurations I_0 and I_1 and two schedules S_0 and S_1 that are applicable to I_0 and I_1, respectively. Consider the set $\mathcal{C}$ of configurations of $\mathcal{A}$ obtained by applying all the prefixes of S_0 and S_1 to, respectively, I_0 and I_1 (line 25).

As in the reduction algorithm $T_{\mathcal{D}\rightarrow\Sigma}$ described in Section 3.2, to determine its next quorum, p uses "fresh" failure detector samples to simulate runs of $\mathcal{A}$ that extend each configuration in $\mathcal{C}$ (lines 29-30). It does so until, for each configuration in $\mathcal{C}$, it has simulated an extension in which it has decided (line 31). The quorum of p is the set of all processes that take steps in these "deciding" extensions (line 32).

Note that in line 27, p waits until it gets a new sample u from its failure detector module (which happens in line 4 of task 1) and then it uses only samples that are more recent than u to extend the configurations in $\mathcal{C}$ (lines 29-30). This ensures the freshness of the failure detector samples that p uses to determine its quorums. Consequently, quorums eventually contain only correct processes (one of the two requirements of Σ).

The proof

Lemma 5.5 *For each correct process p, there is a time after which Σ-output$_p$ contains only correct processes.*

Proof: Let p be a correct process. Note that: (a) p takes a new failure detector sample u infinitely often (in line 27), and (b) Σ-$output_p$ contains only processes that take steps after the most recent sample u taken by p. Since faulty processes eventually stop taking steps, there is a time after which Σ-$output_p$ does not contain any faulty process. □

Lemma 5.6 *For any processes p, q, Σ-$output_p$ and Σ-$output_q$ always intersect.*

Proof: Recall that p and q agreed on a value of the form (I_0, I_1, S_0, S_1) in a (real) execution of $\mathcal{A}$ (line 15). I_0 and I_1 are initial configurations that differ only in the proposal of one process, and S_0 and S_1 are (simulated) schedules of $\mathcal{A}$ in which some process decides 0 in $S_0(I_0)$ and 1 in $S_1(I_1)$. Thus, p and q also agree on the set of configurations $\mathcal{C}$ that is obtained by applying all prefixes of S_0 and S_1 to I_0 and I_1, respectively.

More precisely, let $S_0 = e_1 e_2 \ldots e_\ell$ and $S_1 = f_1 f_2 \ldots f_m$ (where the e_i's and f_j's are steps). Let $C_0 = I_0$ and $C_i = e_i(C_{i-1})$ for $1 \leq i \leq \ell$; similarly, $D_0 = I_1$ and $D_j = f_j(D_{j-1})$ for $1 \leq j \leq m$. Thus, $\mathcal{C} = \{C_0, \ldots, C_\ell, D_0, \ldots, D_m\}$.

Let Q_p and Q_q be the values of Σ-$output_p$ and Σ-$output_q$ at any two times. We must prove that $Q_p \cap Q_q \neq \emptyset$. Suppose, for contradiction, that this is not the case, i.e., $Q_p \cap Q_q = \emptyset$.

Consider the iteration of the loop in lines 26-32 at the end of which p set Σ-$output_p$ to Q_p. In that iteration, for each C_i, $0 \leq i \leq \ell$, p determined a schedule σ_i^p such that p decides some value, denoted x_i^p, in $\sigma_i^p(C_i)$; and, for each D_j, $0 \leq j \leq m$, p determined a schedule τ_j^p such that p decides some value, denoted y_j^p, in $\tau_j^p(C_j)$. Note that Q_p is the set of processes that take steps in some of the schedules σ_i^p, $0 \leq i \leq \ell$, and τ_j^p, $0 \leq j \leq m$.

Consider now the iteration of the loop in lines 26-32 at the end of which q set Σ-$output_q$ to Q_q. We define σ_i^q, x_i^q, τ_j^q, and y_j^q, in an analogous manner. Similarly, Q_q is the set of processes that take steps in the schedules σ_i^q, $0 \leq i \leq \ell$, and τ_j^q, $0 \leq j \leq m$. (See Figure 5.4.)

Claim 5.6.1 *For all i, $0 \leq i \leq \ell$, $x_i^p = x_i^q$; and for all j, $0 \leq j \leq m$, $y_j^p = y_j^q$.*

Proof of Claim 5.6.1. Since Q_p and Q_q are disjoint, for each i, $0 \leq i \leq \ell$, the sets of processes that take steps in σ_i^p and σ_i^q are disjoint. Thus, by Corollary 2.2 (applied with $I = I_0$, $S = e_1 e_2 \ldots e_i$, $S_1 = \sigma_i^p$ and $S_2 = \sigma_i^q$), $x_i^p = x_i^q$. The proof that $y_j^p = y_j^q$ is analogous. □

By Claim 5.6.1, we can now define $x_i = x_i^p = x_i^q$ and $y_j = y_j^p = y_j^q$.

Claim 5.6.2 *For all i, $0 \leq i < \ell$, $x_{i+1} = x_i$; and for all j, $0 \leq j < m$, $y_{j+1} = y_j$.*

Proof of Claim 5.6.2. Recall that $C_{i+1} = e_{i+1}(C_i)$. Since Q_p and Q_q are disjoint, the sets of processes that take steps in σ_i^p and σ_i^q are disjoint. Thus, the process that takes step e_{i+1} does not take a step in at least one of σ_i^p or σ_i^q. Without loss of generality, assume that it does not take a step in σ_i^p. Let $\sigma = e_{i+1} \cdot \sigma_{i+1}^q$. Note that σ is applicable to C_i. Moreover, the sets of processes

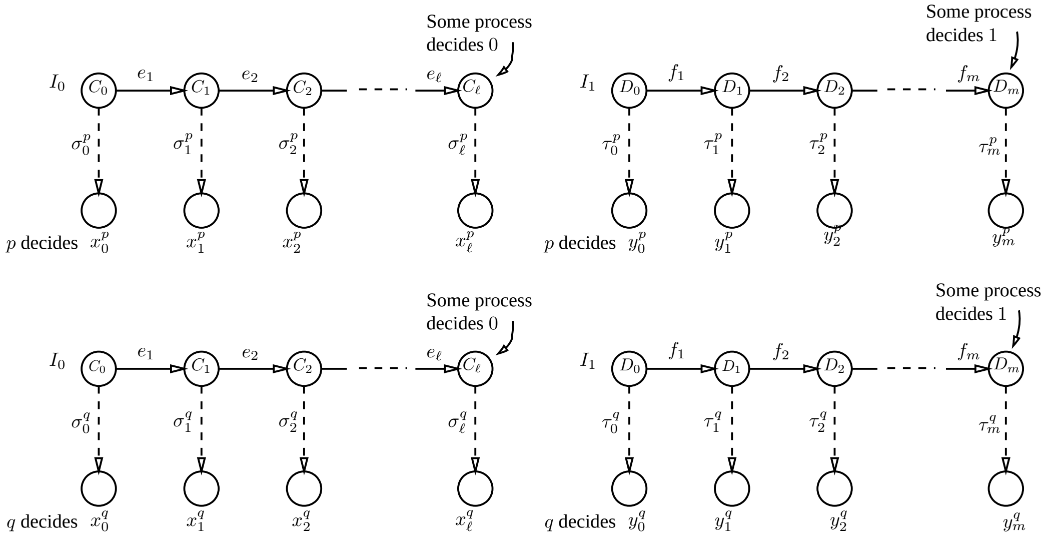

Figure 5.4: Illustration of proof of Lemma 5.6

that take steps in σ^p_i and σ are disjoint. Thus, by Corollary 2.2 (applied with $I = I_0$, $S = e_1 e_2 \ldots e_i$, $S_1 = \sigma^p_i$ and $S_2 = \sigma$), $x_i = x_{i+1}$. The proof that $y_j = y_{j+1}$ is analogous. □

Claim 5.6.3 $x_0 = 0$ *and* $y_0 = 1$.

Proof of Claim 5.6.2. Recall that some process decides 0 in $C_\ell = S_0(I_0)$. Therefore, p decides 0 in $\sigma^p_\ell(C_\ell)$, and so $x_\ell = 0$. By Claim 5.6.2 and a trivial induction, $x_i = 0$ for all i, $0 \leq i \leq \ell$. In particular, $x_0 = 0$. The proof that $y_0 = 1$ is analogous. □

Recall that I_0 and I_1 differ only in the initial value of one process. Let this process be r. Since Q_p and Q_q are disjoint, the sets of processes that take steps in σ^p_0 and τ^q_0 are disjoint. So r does not take a step in at least one of σ^p_0 and τ^q_0. Without loss of generality, assume that r does not take a step in σ^p_0. Thus, σ^p_0 is also applicable to I_1. By Corollary 2.2 (applied with $I = I_1$, S being the empty schedule, $S_1 = \sigma^p_0$ and $S_2 = \tau^q_0$), $x_0 = y_0$. This contradicts Claim 5.6.3, and completes the proof of Lemma 5.6. □

Theorem 5.7 *For all environments $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, then the algorithm in Figure 5.3 transforms $\mathcal{D}$ into Ψ in environment $\mathcal{E}$.*

Proof: Let $\mathcal{A}$ be any algorithm that uses $\mathcal{D}$ to solve QC in environment $\mathcal{E}$. We show that the algorithm in Figure 5.3 uses $\mathcal{A}$ to transform $\mathcal{D}$ into Ψ in environment $\mathcal{E}$. In that algorithm, each process p maintains a variable Ψ-*output*$_p$. We now prove that the values that these variables take conform to the specification of Ψ. By inspection of Figure 5.3, it is clear that Ψ-*output*$_p$ is either $\perp$, or **red**

(in which case we say it is of type $\mathcal{FS}$), or a pair (q, Q) where $q \in \Pi$ and $Q \subseteq \Pi$ (in which case we say it is of type (Ω, Σ)).

(1) *For each process p, Ψ-output$_p$ is initially $\bot$ (line 1). If Ψ-output$_p$ ever changes value, it becomes of type $\mathcal{FS}$ forever (line 18) or of type (Ω, Σ) forever (lines 20-34).*

(2) *For all processes p and q, it is impossible for Ψ-output$_p$ to be of type $\mathcal{FS}$ and Ψ-output$_q$ to be of type (Ω, Σ).* The types of Ψ-*output*$_p$ and Ψ-*output*$_q$ are determined by the value returned by QC algorithm $\mathcal{A}$ in line 15. By the Agreement property of QC, p and q cannot decide different values in line 15.

(3) *For each correct process p, eventually Ψ-output$_p \neq \bot$.* To see this, let p be any correct process. Process p simulates a forest Υ_p of ever-increasing partial runs of $\mathcal{A}$ as in [14] (see line 6). In this simulation, every tree in Υ_p has runs in which all correct processes take steps infinitely often and receive all messages sent to them. So, by the Termination property of QC, every tree in Υ_p has a run in which p decides. Therefore, eventually *all* correct processes complete the wait statement in line 8, and execute $\mathcal{A}$ in line 11 or 14. By the Termination property of QC, eventually p decides in that execution of $\mathcal{A}$, and stops waiting in line 15. Thus, p eventually sets Ψ-*output*$_p$ to a value other than $\bot$ in line 18 or 34.

(4) *For each process p, if Ψ-output$_p$ is* **red** *then a process previously crashed in the current run.* To see this, let p be some process that sets Ψ-*output*$_p$ = **red** (line 18). Thus, p decides 0 or $\mathcal{Q}$ in the execution of $\mathcal{A}$ that it invoked in line 11 or 14. If p decides $\mathcal{Q}$ then the fact that some process has previously crashed in the current run follows immediately from part (b) of the Validity property of QC. If p decides 0 then from part (a) of the Validity property of QC, some process q proposed 0 in the execution of $\mathcal{A}$ that q invoked in line 11. This implies that q decided $\mathcal{Q}$ in one of the simulated runs of $\mathcal{A}$ that q has in its forest Υ_q. Recall that these are runs that could have occurred with the current failure pattern. By part (b) of the Validity property of QC, this means that some process has previously crashed in the current run.

(5) *If the Ψ-output variable of any process is ever of type (Ω, Σ), then there is a time after which, for every correct process p, Ω-output$_p$ is the id of the same correct process.* To see this, suppose some Ψ-*output* becomes of type (Ω, Σ). Then, by (2) and (3) above, eventually the Ψ-*output* variable of every correct process also becomes of type (Ω, Σ). So every correct process sets its Ω-*output* variable repeatedly in line 22 using the extraction procedure described in [14]. Since processes reach line 22, by Lemma 5.4, a critical index exists in the limit forest Υ. By following the proof of [14], it can now be shown that eventually all the correct processes extract the id of the same correct process. The only difference is that whenever [14] refers to a *bi*valent node, we now refer to a *multi*valent one, and whenever [14] refers to 0-valent versus 1-valent nodes, we refer here to u-valent and v-valent nodes where $u, v \in \{0, 1, \mathcal{Q}\}$ and $u \neq v$.

(6) *If the Ψ-output variable of any process is ever of type (Ω, Σ) then: (a) for every correct process p, there is a time after which Σ-output$_p$ contains only correct*

processes, and (b) for every processes p and q, Σ-$output_p$ and Σ-$output_q$ always intersect. This is shown in Lemmas 5.5 and 5.6.

From the above, it is clear that the values of the variables Ψ-*output* conform to Ψ: For an initial period of time they are equal to $\perp$. Eventually, however, they behave either like the failure detector (Ω, Σ) at all processes or, if a failure occurs, they may instead behave like the failure detector $\mathcal{FS}$ at all processes. Moreover, this switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ is consistent at all processes. □

From Theorems 5.2 and 5.7, we have:

Corollary 5.8 *For all environments $\mathcal{E}$, Ψ is the weakest failure detector to solve QC in $\mathcal{E}$.*

5.3 The weakest failure detector to solve NBAC

5.3.1 Specification of NBAC

In the *non-blocking atomic commit* problem (NBAC), each process p starts with a value $v \in \{\text{Yes}, \text{No}\}$ (we say that *p votes v*), and terminates with a *decision* value Commit or Abort. It is required that:

Termination: If every correct process votes, then every correct process eventually returns a value.

Agreement: No two processes (whether correct or faulty) return different values.

Validity: A process may only return Commit or Abort. Moreover,
(a) If $v =$ Commit then all processes previously voted Yes.
(b) If $v =$ Abort then either some process previously voted No or a failure previously occurred.

Despite their apparent similarity, QC and NBAC are different in important ways. In NBAC, the two possible input values Yes and No are not symmetric: A single vote of No is enough to force the decision to abort. In contrast, in QC, (as in consensus) no input value has a privileged role. Another way in which the two problems differ is that the semantics of the decision to abort (in NBAC) and the decision to quit (in QC) are different. In NBAC, the decision to abort is sometimes inevitable (e.g., if a process crashes before voting); in contrast, in QC, the decision to quit is never inevitable, it is only an option. Moreover, in NBAC, the decision to abort signifies that either a failure has occurred *or* someone voted No; in contrast, in QC, the decision to quit is allowed only if a failure has occurred.

The following corollary to Lemma 2.3 follows directly from the specification of NBAC:

Corollary 5.9 *Let $\mathcal{A}$ be any NBAC algorithm, $R = \langle F, H, I, S, T \rangle$ be any partial run of $\mathcal{A}$, p and q be any processes such that no step of q causally precedes the last step of p in R. Then if p decides v in R, then $v =$ Abort.*

5.3.2 Using $\mathcal{FS}$ to relate NBAC and QC

We first show that NBAC is equivalent to the combination of QC and failure detector $\mathcal{FS}$. We then use this result to establish a relationship between the weakest failure detector to solve QC and the one to solve NBAC.

Theorem 5.10 *NBAC is equivalent to QC and $\mathcal{FS}$. That is, in every environment $\mathcal{E}$:*
(a) Given failure detector $\mathcal{FS}$, any solution to QC can be transformed into a solution to NBAC.
(b) Any solution to NBAC can be transformed into a solution to QC, and can be used to implement $\mathcal{FS}$.

Proof: Let $\mathcal{E}$ be an arbitrary environment.

(a) The algorithm in Figure 5.5 uses $\mathcal{FS}$ to transform QC into NBAC in $\mathcal{E}$. Each process p sends its vote to all processes and then waits until the votes of all processes are received or $\mathcal{FS}$ detects a failure by outputting **red**. If the votes of all processes are received and are Yes, then p runs a QC algorithm proposing 1. Otherwise (i.e., if some vote was No or a failure was detected by $\mathcal{FS}$), p runs a QC algorithm proposing 0.

If 1 is decided in the QC algorithm, then p returns Commit. If 0 or $\mathcal{Q}$ is decided, then p returns Abort.

The Agreement property of QC ensures that no two processes decide differently. If there are no no failures, then eventually p receives all the votes. If a failure occurs, then $\mathcal{FS}$ eventually outputs **red**. Hence, the wait statement in line 2 is non-blocking. The Termination property of QC ensures that every correct process eventually decides.

Assume that p decides Commit. By Validity of QC some process q previously proposed 1. By the algorithm, q received the votes of all processes and all the votes were Yes.

Assume now that p decides Abort. By Validity of QC some process q previously proposed 0 or a failure previously occurred. If some process q proposed 0, then either q received vote No from some process or a failure previously occurred and was detected by $\mathcal{FS}$.

In both cases, Validity of NBAC is ensured.

(b) It is known that NBAC can be used to implement $\mathcal{FS}$ in any environment [16, 32]. Roughly speaking, processes use the given NBAC algorithm repeatedly (forever), voting Yes in each instance. At each process, the output of $\mathcal{FS}$ is initially **green**, and becomes permanently **red** if and when an instance of NBAC returns Abort. It remains to prove that any solution to NBAC in $\mathcal{E}$ can be transformed into a solution to QC in $\mathcal{E}$. Such a transformation is shown in Figure 5.6.

Each process p sends its proposal to all and executes the given NBAC algorithm $\mathcal{A}$ by voting Yes. If $\mathcal{A}$ returns Abort, then p returns $\mathcal{Q}$. If $\mathcal{A}$ returns Commit, then p waits until it receives the proposals of all processes, and returns the smallest proposal among them.

```
Procedure nbacVote($v$): { $v$ is Yes or No }
  send $v$ to all
  wait until [(for each process $q$ in $\Pi$, received $q$'s vote) or $\mathcal{FS} = \textbf{red}$]
  if the votes of all processes are received and are Yes then
    $myproposal \leftarrow 1$
  else { some vote was No or there was a failure }
    $myproposal \leftarrow 0$
  $mydecision \leftarrow$ qcPropose($myproposal$) { run the given QC algorithm }
  if $mydecision = 1$ then
    return Commit
  else { $mydecision = 0$ or $\mathcal{Q}$ }
    return Abort
```

Figure 5.5: Using $\mathcal{FS}$ to transform QC into NBAC: code for each process p

```
Procedure qcPropose($v$): { $v$ is 1 or 0 }
  send $v$ to all
  $d :=$ nbacVote(Yes) { run the given NBAC algorithm $\mathcal{A}$ }
  if $d =$ Abort then
    return $\mathcal{Q}$
  else
    wait until [for each process $q \in \Pi$, received $q$'s proposal]
    return smallest proposal received
```

Figure 5.6: Transforming NBAC into QC: code for each process p

By the Termination property of NBAC, $\mathcal{A}$ eventually returns a value in $\{\text{Commit}, \text{Abort}\}$ at every correct process. By the Agreement property of NBAC, no two different values are returned by $\mathcal{A}$ at any two processes.

Assume that $\mathcal{A}$ returns Abort. By the algorithm in Figure 5.6, every process returns $\mathcal{Q}$ or crashes — the Agreement and Termination properties of QC are satisfied. By the Validity (b) property of NBAC and the fact that no process votes No in its execution of $\mathcal{A}$, it follows that a failure previously occurred — the Validity property of QC is satisfied.

Assume now that $\mathcal{A}$ returns Commit. By the algorithm in Figure 5.6, every process p waits until it receives the proposals of all processes, and then returns the smallest proposal received. Let R be a partial run of $\mathcal{A}$ such that p decides Commit in its last step in R. Corollary 5.9 implies that the last of step of p in R is causally preceded by at least one step of every process q. By the algorithm in Figure 5.6, before taking any step in R, q has previously sent its proposal to all. Thus, eventually, p either crashes, or receives the proposals of all processes and returns the smallest among them — the Agreement, Termination and Validity properties of QC are satisfied. □

5.3.3 The weakest failure detector to solve NBAC

Theorem 5.11 *For every environment $\mathcal{E}$, if $\mathcal{D}$ is the weakest failure detector to solve QC in $\mathcal{E}$, then $(\mathcal{D}, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

Proof: Let $\mathcal{E}$ be an arbitrary environment, and $\mathcal{D}$ be the weakest failure detector to solve QC in $\mathcal{E}$. This means that: (i) $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$ and (ii) any failure detector that solves QC in $\mathcal{E}$ can be transformed into $\mathcal{D}$ in $\mathcal{E}$.

Let $\mathcal{D}' = (\mathcal{D}, \mathcal{FS})$. We must show that: (a) $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$, and (b) any failure detector that solves NBAC in $\mathcal{E}$ can be transformed into $\mathcal{D}'$ in $\mathcal{E}$.

(a) Since the output of $\mathcal{D}'$ includes the output of $\mathcal{D}$, by (i), $\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$. Since $\mathcal{D}'$ also includes $\mathcal{FS}$, by Theorem 5.10(a), $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$.

(b) Let $\mathcal{D}''$ be a failure detector that solves NBAC in $\mathcal{E}$. By Theorem 5.10(b), (1) $\mathcal{D}''$ can be used to solve QC in $\mathcal{E}$, and (2) $\mathcal{D}''$ can be used to implement $\mathcal{FS}$ in $\mathcal{E}$. From (1) and (ii), $\mathcal{D}''$ can be transformed into $\mathcal{D}$ in $\mathcal{E}$. By (2), $\mathcal{D}''$ can be transformed into $(\mathcal{D}, \mathcal{FS})$, i.e., into $\mathcal{D}'$, in $\mathcal{E}$. □

From Corollary 5.8 and Theorem 5.11, we immediately have:

Corollary 5.12 *For all environments $\mathcal{E}$, $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

5.4 Concluding remarks

Handling future failures

Our definitions of QC and NBAC do not allow a process to quit or abort because of a future failure. We could have defined these problems in a way that allows such behavior, as in fact is the case in some specifications of NBAC in the literature [31, 68]. Our results also hold with these definitions, provided we make a corresponding change to the definitions of the failure detectors $\mathcal{FS}$ and Ψ: they are now allowed to output **red** in executions with failures even *before* a failure has occurred.

Environments with a majority of correct processes

In environments where a majority of processes are correct it is easy to implement the quorum failure detector Σ: Each process periodically sends "join-quorum" messages, and takes as its present quorum any majority of processes that respond to that message [20]. Therefore, in such environments Ψ is equivalent to a simpler failure detector, one which outputs just Ω where Ψ outputs (Ω, Σ).

Other models

The solutions presented in Figures 5.2 and 5.5 are also correct in a weaker model where steps of algorithms have finer granularity (send phase, query phase, and receive phase are not encapsulated in the same atomic step), and channels guarantee only that every correct process eventually receives every message sent to it by any *correct* process.

The algorithm in Figure 5.6 can be easily adapted for the weaker model. In executing the NBAC algorithm $\mathcal{A}$, each process collects the set of proposal values of all processes whose steps in the current run of $\mathcal{A}$ causally precede p's decision. This set is determined by having p tag every message m (sent in the context of the current execution of $\mathcal{A}$) with the union of proposals attached to all messages m' causally preceding m (including its own proposal). If $\mathcal{A}$ returns Commit at p, then, by Corollary 5.9, p collects the proposals of all processes. In this case, p returns the smallest proposal among them.

Similarly, the algorithm of Figure 5.1 can also be adapted for the weaker model by forwarding the proposal value of $p_{(k \mod n)+1}$ as an attachment to every message sent in the context of each k-th instance of the binary QC algorithm in which 1 is proposed.

The reduction algorithm described in Figure 5.3 is correct also in a stronger model in which every process can atomically send its messages to all.

5.5 Related work

NBAC has been studied extensively in the context of transaction processing [31, 68]. Its relation to consensus was first explored in [37]. The timing conditions sufficient for solving NBAC in a probabilistic manner were explored in [18]. Charron-Bost and Toueg [16] and Guerraoui [32] showed that despite some apparent similarities, in asynchronous systems NBAC and consensus are in general incomparable — i.e., a solution for one problem cannot be used to solve the other. An exception is the case where at most one process may fail. In this case, NBAC can be transformed into consensus, but the reverse does not hold [16, 32].

The problem of determining the weakest failure detector to solve NBAC was explored and settled in special settings. Fromentin *et al.* [30] showed that to solve NBAC between *every* pair of processes in the system, one needs a perfect failure detector $\mathcal{P}$. Guerraoui and Kouznetsov [33] determined the weakest failure detector for NBAC within a restricted class of failure detectors, while from results of [16] and [32] it follows that in the special case where at most one process may crash, $\mathcal{FS}$ is the weakest failure detector to solve NBAC.

Guerraoui and Kouznetsov directly determined the weakest failure detector for solving NBAC with a majority of correct processes [35]. Independently, Hadzilacos and Toueg obtained the same result by introducing the quittable consensus problem (QC), determining the weakest failure detector for solving QC with a majority of correct processes, and establishing the relationship between QC and NBAC [40].

A problem similar to QC, called *detectable agreement*, was introduced by Fitzi *et al.* [29] for Byzantine environments. In detectable Byzantine agreement, correct processes can sometimes agree on the fact that some of the processes misbehave which matches the specification of QC for the case of crash failures.

Chapter 6

Failure Detectors as Type Boosters

In this chapter, we show that Ω_n is the weakest failure detector to solve consensus among k processes using registers and objects of any *m-ported one-shot deterministic* type T such that $m \leq n+1$ and *cons*$(T) = n$. As a corollary of our result, we show that Ω_{t+1} is the weakest failure detector to solve consensus among k processes using registers and *t-resilient* objects of any types (not necessarily one-shot deterministic types with a bounded number of ports).

Section 6.1 presents necessary details on the model used in this chapter. Section 6.2 recalls the hierarchy of failure detectors Ω_n. Section 6.3 shows that Ω_n is necessary to boost the consensus power of one-shot deterministic objects one level up in the consensus hierarchy. Section 6.4 generalizes this result to show that Ω_n is necessary to boost the consensus power of at most *$(n+1)$-ported* one-shot deterministic objects to any higher level in the consensus hierarchy. Section 6.5 applies our result to the question of boosting the resilience of a distributed system. Section 6.7 concludes the chapter by discussing the related work.

An early version of the results of this chapter appeared in [34].

6.1 Preliminaries

We consider a set Π of k asynchronous processes $p_1, p_2, \ldots, p_k$ ($k \geq 2$) that communicate using shared objects. The processes may fail by crashing, i.e., stop executing their steps. A process that never crashes is said to be *correct*. A process that is not correct is said to be *faulty*. We say that a subset $U \subseteq \Pi$ is *alive* if U includes at least one correct process. We consider here all failure patterns, i.e., we make no assumptions on when and where failures might occur.

6.1.1 Objects and types

Let for any $m \in \mathbb{N}$, $\mathbb{N}_m = \{1, \ldots, m\}$. An *object* is a data structure that can be accessed concurrently by the processes. Every object is an instance of a sequential *type* which is defined by a tuple (m, O, R, Q, δ). Here, m is a positive integer denoting the number of *ports* (corresponding to the maximum number of processes that can concurrently access an object of this type), O is a set of *operations*, R is a set of *responses*, Q is a set of *states*, and δ is a relation known as the *sequential specification* of the type: it carries each state, operation and port number to a set of response and state pairs. We say that a type (m, O, R, Q, δ) is *deterministic* if its sequential specification is a function $\delta : Q \times O \times \mathbb{N}_m \rightarrow Q \times R$. In most of this chapter, we assume that object types are deterministic. A type with m ports is said to be *m-ported.*

Informally, a port of an object of a *one-shot* type can be consistently used at most once. More precisely, a type (m, O, R, Q, δ) is one-shot if $R = R' \cup \{\bot\}$, $Q = Q' \times 2^{\mathbb{N}_m}$, and for all $o \in O$, $q, q' \in Q'$, $U, U' \in 2^{\mathbb{N}_m}$, $j \in \mathbb{N}_m$, and $r \in R$:

$$\begin{aligned} &((q', U'), r) \in \delta((q, U), o, j) \quad \Rightarrow \\ &\Big((j \notin U) \wedge (U' = U \cup \{j\}) \wedge (r \in R')\Big) \vee \\ &\Big((j \in U) \wedge (q' = q) \wedge (U' = U) \wedge (r = \bot)\Big) \end{aligned}$$

The $\mathsf{register}$ type is defined as a tuple (k, O, R, Q, δ) where Q is a set of *values* that can be stored in a register, $O = \{\mathit{read}(), \mathrm{write}(v) : v \in Q\}$, $R = Q \cup \{\mathit{ok}\}$ and $\forall v, v' \in Q$ and $j \in \mathbb{N}_k$, $\delta(v, \mathit{write}(v'), j) = (v', \mathit{ok})$ and $\delta(v, \mathit{read}(), j) = (v, v)$.[1]

A process accesses objects by invoking operations on the ports of the objects. Unless explicitly stated otherwise, we assume that, in any execution, a port can be used by at most one process, and a process can use at most one port of each object (this corresponds to the *one-to-one static binding scheme* [13]). In Section 6.5, we also consider the more permissive *softwired* binding scheme [13].

We consider here *linearizable* [43] objects: even though operations of concurrent processes may overlap, each operation takes effect instantaneously between its invocation and response. If a process invokes an operation on a linearizable object and fails before receiving a matching response, then the "failed" operation *may* take effect at any time after the corresponding invocation. Any execution on linearizable objects can thus be seen as a sequence of atomic invocation-response pairs.

Unless explicitly stated otherwise, we assume that the object implementations are *wait-free*: any object operation invoked by a correct process eventually returns, regardless of failures of other processes [41]. In contrast, *t-resilient* implementations of shared objects (considered in Section 6.5) provide weaker liveness properties, they only guarantee that a correct process completes its operation, as long as no more than t processes crash.

6.1.2 Algorithms

We define an *algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* as a collection of k deterministic automata, one for each process in the system. $\mathcal{A}(p)$ denotes the automaton on which process p runs the algorithm $\mathcal{A}$. Computation proceeds in atomic *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p:

(i) performs an operation on a shared object *or* queries its failure detector module $\mathcal{D}_p$, and

[1] Note that our definition of the $\mathsf{register}$ type causes no loss of generality in describing the read/write shared memory [71, 44].

(ii) applies its current state together with the response received from a shared object *or* the value received from $\mathcal{D}_p$, during that step, to the automaton $\mathcal{A}(p)$ to obtain a new state.

A step of $\mathcal{A}$ is thus identified by a pair (p, x), where x is either λ (the empty value) or the failure detector value output at p during that step.

6.1.3 Configurations, schedules and runs

A *configuration* of $\mathcal{A}$ defines the current state of each process and each object in the system. An *initial configuration* of $\mathcal{A}$ defines the initial state of each process p and each object.

Let C be any configuration of $\mathcal{A}$. For every process p, $\mathcal{A}(p)$ and the state of p in C determine whether, in any step of p applied to C, p *accesses* a shared object X (we say that *p accesses X in C*) or queries its failure detector module (we say that *p queries $\mathcal{D}$ in C*). A step (p, x) is said to be *applicable* to C if:

(a) $x = \lambda$, and p accesses some shared object X in C, or

(b) $x \in \mathcal{R}_{\mathcal{D}}$, and p queries $\mathcal{D}$ in C.

In case (b), x is interpreted as the value output by $\mathcal{D}_p$ during step (p, x). For a step e applicable to C, $e(C)$ denotes the unique configuration that results from applying e to C (the configuration is unique because algorithms and objects considered here are deterministic).

A *schedule* S of algorithm $\mathcal{A}$ is a (finite or infinite) sequence of steps of $\mathcal{A}$. $S_\perp$ denotes the empty schedule. We say that *a schedule S is applicable to a configuration C* if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule S applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C.

Let S be any schedule applicable to a configuration C, and X be any object. We say that *S applied to C accesses X* if S has a prefix $S' \cdot (p, \lambda)$ where p accesses X in $S'(C)$.

For any $P \subseteq \Pi$, we say that S is a *P-solo schedule* if only processes in P take steps in S.

A *partial run of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T \rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is a *finite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is a *finite* list of increasing time values such that $|S| = |T|$, S is applicable to I, and for all $1 \leq k \leq |S|$, if $S[k] = (p, x)$ then:

(1) Either p has not crashed by time $T[k]$, i.e., $p \notin F(T[k])$, or $x = \lambda$ and $S[k]$ is the last appearance of p in S, i.e., $\forall k < k' \leq |S|$: $S[k'] \neq (p, *)$ [2];

(2) if $x \in \mathcal{R}_{\mathcal{D}}$, then x is the value of the failure detector module of p at time $T[k]$, i.e., $x = H_{\mathcal{D}}(p, T[k])$.

[2] The last condition takes care about the cases when an operation of p is linearized *after* p has crashed, and there can be at most one such operation in a run.

A *run of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of $\mathcal{A}$, S is an *infinite* schedule of $\mathcal{A}$, and $T \subseteq \mathbb{T}$ is an *infinite* list of increasing time values indicating when each step of S has occurred. In addition to satisfying properties (1) and (2) of a partial run, R should guarantee that:

(3) every process that is correct (in F) takes an infinite number of steps in S.

6.1.4 Consensus and consensus power

Let $\mathcal{S}$ be any set of types and $\mathcal{D}$ be any failure detector. We say that an algorithm $\mathcal{A}$ solves consensus using $\mathcal{D}$ and objects of types in $\mathcal{S}$ if, in the initial configurations of $\mathcal{A}$, processes *propose* values in $\{0, 1\}$, and in every run of $\mathcal{A}$, only $\mathcal{D}$ and objects of types in $\mathcal{S}$ are accessed and the Termination, Agreement, and Validity properties (Section 2.7) are satisfied.

We say that a set $\mathcal{S}$ of types *solves m-process consensus* if there is an algorithm for m processes that solves consensus using *only* registers and objects of types in $\mathcal{S}$ (without failure detectors).

The *consensus power* of an object type T, denoted $\mathit{cons}(T)$, is the largest number m of processes such that $\{T\}$ solves m-process consensus. If no such largest m exists, then $\mathit{cons}(T) = \infty$.

It is sometimes convenient to think of the consensus problem in terms of an object type. Formally, the m-process consensus type is specified as a tuple (m, O, R, Q, δ), where $Q = \{\bot, 0, 1\} \times 2^{\mathbb{N}_m}$, $O = \{\mathit{propose}(v) : v \in \{0, 1\}\}$, $R = \{\bot, 0, 1\}$, and $\forall v, v' \in \{0, 1\}$, $U \in 2^{\mathbb{N}_m}$, and $j \in \mathbb{N}_m$, if $j \notin U$, then $\delta((\bot, U), \mathit{propose}(v), j) = ((v, U \cup \{j\}), v)$ and $\delta((v', U), \mathit{propose}(v), j) = ((v', U \cup \{j\}), v')$, otherwise (if $j \in U$), $\delta((v', U), \mathit{propose}(v), j) = ((v', U), \bot)$. Clearly, the m-process consensus type is m-ported, one-shot, and deterministic.

6.1.5 Team consensus

We also use a restricted form of consensus, called *team consensus*. This variant of consensus always ensures Validity and Termination, but Agreement is ensured only if the input values satisfy certain conditions. More precisely, assume that there exists a (known a priori) partition of the processes into two non-empty sets (teams). Team consensus ensures Validity, Termination and

Team Agreement: If all processes in a team have the same input value, then no two processes decide different values.

Obviously, team consensus can be solved whenever consensus can be solved. Surprisingly, the converse is also true [61, 64]:

Lemma 6.1 *Let $\mathcal{S}$ be any set of types. If team consensus among $m \geq 2$ processes can be solved using objects of types in $\mathcal{S}$ (without failure detectors), then $\mathcal{S}$ also solves m-process consensus.*

6.1.6 Weak consensus

To prove our result, we also consider a weaker form of consensus, the *weak consensus* problem [28]. Recall that weak consensus ensures the Termination and Agreement properties of consensus, but Validity is replaced by a weaker Non-Triviality property: every algorithm that solves the weak consensus problem has a run in which 0 is decided and a run in which 1 is decided. Obviously, weak consensus can be solved whenever consensus can be solved. We show below that, if deterministic types are used, the converse is also true:

Lemma 6.2 *Let $\mathcal{S}$ be any set of deterministic types. If weak consensus among $m \geq 2$ processes can be solved using objects of types in $\mathcal{S}$ (without failure detectors), then $\mathcal{S}$ solves m-process consensus.*

Proof: Let $\mathcal{A}$ be any algorithm that solves weak consensus among $m \geq 2$ processes using objects of types in $\mathcal{S}$.

Let G be the *execution graph* of $\mathcal{A}$: the vertices of G are all possible configurations of $\mathcal{A}$ (defined by the states of the processes and all shared objects); and vertices s, s' of G are connected with an edge directed from s to s' if and only if there is a step of $\mathcal{A}$ that, applied to s, results in s'.

A vertex s of G is assigned a tag $v \in \{0, 1\}$ if it has a descendant s' in G (i.e., there exists a path in G from s to s') such that some process has decided v in s'. If a configuration has both tags 0 and 1, it is called *bivalent.* If a configuration has only one tag v, it is called *v-valent.* A configuration is univalent if it is 0-valent or 1-valent. The Termination property of weak consensus ensures that any configuration of $\mathcal{A}$ is either bivalent or univalent. It is straightforward to show that $\mathcal{A}$ has a bivalent initial configuration [28].

We show first that there exists a *critical* configuration in G, i.e., a bivalent configuration $\bar{s}$ such that every step of $\mathcal{A}$ applied to $\bar{s}$ results in a univalent configuration. Suppose not, i.e., every bivalent configuration in G has a bivalent descendant. Then, starting from any bivalent initial configuration of $\mathcal{A}$, we can build an infinite schedule of step of $\mathcal{A}$ that goes through bivalent configurations only. By the Agreement property of weak consensus, no process can decide in a bivalent configuration. Since we consider here all failure patterns, this schedule corresponds to a run of $\mathcal{A}$ in which no process can ever decide — a contradiction with the Termination property of weak consensus.

Assume now that the system is in a critical configuration $\bar{s}$. Since the algorithm $\mathcal{A}$ and the types in $\mathcal{S}$ are deterministic, the step of any given process applied to $\bar{s}$ triggers exactly one transition in graph G. Thus, for any step of $\mathcal{A}$ applied to $\bar{s}$, the valence of the resulting configuration is defined by the identity of the process that takes that step. Now we partition Π into two teams Π_0 and Π_1: for each $i \in \{0, 1\}$, Π_i consists of all processes whose steps applied to $\bar{s}$ result in i-valent configuration. Since $\bar{s}$ is bivalent and any step applied to $\bar{s}$ results in a univalent configuration, the two teams are non-empty. The algorithm in Figure 6.1 solves team consensus problem team consensus among m processes for teams Π_0 and Π_1.

Let all objects used by $\mathcal{A}$ be initialized to their states in $\bar{s}$. For each $i \in \{0, 1\}$, we associate team Π_i with a register X_i. Every process p writes its input value

```
Initially:
    all objects are initialized to their states in $\bar{s}$

Procedure tcPropose($v$):    { let $p \in \Pi_i$, $i \in \{0,1\}$ }
    $X_i \leftarrow v$    { write the proposal in the team's register }
    let $p$ be initialized to its state in $\bar{s}$
    run $\mathcal{A}$ until it returns a value $j \in \{0,1\}$
    return $X_j$
```

Figure 6.1: A team consensus algorithm: code for each process p

into its team's register and then runs $\mathcal{A}$ starting from p's state in $\bar{s}$ until $\mathcal{A}$ returns a value $j \in \{0,1\}$. Then the process returns the value of X_j.

Consider any run R of the algorithm in Figure 6.1. The Termination property of weak consensus ensures Termination of our algorithm. Assume that p returns a value of X_j in R, i.e. $\mathcal{A}$ returns j at p. By the definition of Π_0 and Π_1, the first step accessing X in R is by a process $q \in \Pi_j$. By the algorithm, q has previously written its input value in X_j. Thus, Validity of team consensus is ensured. The Agreement property of weak consensus ensures that $\mathcal{A}$ cannot return $1-j$ at any process q in R. Thus, no process can return a value of X_{1-j} in R. Assume now that all processes in a team propose the same value. Hence, Agreement is ensured, since the processes return the value previously written in X_j, and no two different values can be written in X_j.

Thus, team consensus can be solved among m processes using objects of types in $\mathcal{S}$ initialized to their states in $\bar{s}$. By Lemma 6.1, $\mathcal{S}$ solves consensus among m processes. □

6.2 Hierarchy of failure detectors Ω_n

The hierarchy of failure detectors Ω_n ($n \in \mathbb{N}$) was introduced in [61]. Ω_n ($n \in \mathbb{N}$) outputs a set of *at most* n processes at each process so that, eventually, the same *alive* set (including at least one correct process) is output at all correct processes.

Formally, $\mathcal{R}_{\Omega_n} = \{P \subseteq \Pi : |P| \leq n\}$, and for each failure pattern F:

$$\begin{aligned} &H \in \Omega_n(F) \quad \Rightarrow \quad \exists t \in \mathbb{T}\ \exists P \in \mathcal{R}_{\Omega_n}\ \forall p \in \mathit{correct}(F)\ \forall t' \geq t\ : \\ &\quad (P \cap \mathit{correct}(F) \neq \emptyset) \wedge (H(p,t') = P) \end{aligned}$$

Note that Ω_1 is equivalent to Ω. It was shown in [61] that, for all $n \geq 1$:

(a) $\Omega_{n+1} \prec \Omega_n$;

(b) for any type T such that $\mathit{cons}(T) = n$, and any $k > n$, there is an algorithm that solves k-process consensus using Ω_n, registers, and objects of type T.

6.3 Boosting consensus power to level $n+1$

In this section, we assume that $k = n+1$ processes communicate through registers and objects of a one-shot deterministic type T such that $\mathit{cons}(T) \leq n$. We show that Ω_n is necessary to solve consensus in this system. Our proof is a generalization of the proof that Ω is necessary to solve consensus in message-passing asynchronous systems [14] (see Section 3.1).

6.3.1 Overview of the reduction algorithm

Let $n \geq 1$ and $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus among $k = n + 1$ processes using a failure detector $\mathcal{D}$, registers and objects of a one-shot deterministic type T such that $cons(T) \leq n$. We describe a reduction algorithm $T_{\mathcal{D}\to\Omega_n}$ that emulates the output of Ω_n using $\mathcal{D}$ and $\mathsf{Cons}_{\mathcal{D}}$. The reduction algorithm has all correct processes eventually agree on the same *alive* set of at most n processes.

Like the reduction algorithm of Section 3.1, $T_{\mathcal{D}\to\Omega_n}$ consists of two parallel tasks: a communication task and a computation task.

In the communication task, each process p periodically queries its failure detector module of $\mathcal{D}$ and exchanges the failure detector values with the other processes values using read-write memory. All this information is pieced together in a directed acyclic graph (DAG) G_p.

Let I^l ($l = 0, 1, \dots, k$) denote an initial configuration of $\mathsf{Cons}_{\mathcal{D}}$ in which processes $p_1, \dots, p_l$ propose 1 and processes $p_{l+1}, \dots, p_k$ propose 0. In the computation task, p periodically uses its DAG G_p to simulate *locally*, for every initial configuration I^l ($l = 0, 1, \dots, k$) and every set of processes $P \subseteq \Pi$, a number of finite P-solo schedules of runs of $\mathsf{Cons}_{\mathcal{D}}$ that *could have occurred* with the current failure pattern and failure detector history. These schedules constitute an ever-growing *simulation tree*, denoted $\Upsilon_p^{P,l}$. Since registers provide reliable communication, all such $\Upsilon_p^{P,l}$ tend to the same *infinite* simulation tree $\Upsilon^{P,l}$ [14].

It turns out that the processes can eventually detect the same set $P \subseteq \Pi$ such that P includes all correct processes, and either (a) there exists a correct *critical* process whose proposal value in some initial configuration I^l determines the decision value in all paths in $\Upsilon^{P,l}$, or (b) some $\Upsilon^{P,l}$ has a finite subtree γ, called a *complete decision gadget*, that provides sufficient information to compute a set of at most n processes at least one of which is correct, called the *deciding set* of γ. Eventually, the correct processes either detect the same critical process or compute the same complete decision gadget and agree on its deciding set. In both cases, Ω_n is emulated.

A difficult point here is that, sometimes, the deciding set is encoded in an object of type T. Since the specification of T is a parameter of the reduction algorithm, we cannot directly use the case analysis of [14] (Section 3.1) to compute the deciding set. Fortunately, in this case, using the fact that type T is one-shot and deterministic, we can locate a special kind of decision gadget, which we introduce here and which we call a *rake*. Informally, a rake can be used to identify a set of "confused" processes that are not able to decide without communicating with the other processes. Thus, the set of "non-confused" processes (of size at most n) must contain at least one correct process.

```
Initially:
    $G_p \leftarrow$ empty graph
    $k_p \leftarrow 0$

while $true$ do
    for all $q \in \Pi$ do $G_p \leftarrow G_p \cup G_q$
    $d_p \leftarrow$ query failure detector $\mathcal{D}$
    $k_p \leftarrow k_p + 1$
    add to $G_p$ $[p, d_p, k_p]$ and edges from all vertices of $G_p$ to $[p, d_p, k_p]$
```

Figure 6.2: Building a DAG: code for each process p

6.3.2 DAGs and simulation trees

The communication task of algorithm $T_{\mathcal{D}\rightarrow\Omega}$ is presented in Figure 6.2. This task maintains an ever-growing DAG that satisfies properties (1)–(4) of Section 3.1.2. For each process p, the DAG is stored in a register G_p which is periodically updated by p and read by all processes.

Let I^l ($l = 0, 1, \ldots, k$) denote an initial configuration of $\mathsf{Cons}_{\mathcal{D}}$ in which processes $p_1, \ldots, p_l$ propose 1 and processes $p_{l+1}, \ldots, p_k$ propose 0. Let $P \subseteq \Pi$ be any set of processes, and $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots [q_s, d_s, k_s]$ be any path in G_p such that $\forall i \in \{1, 2, \ldots, s\} : q_i \in P$. Since algorithms and shared objects considered here are deterministic, g and I^l *induce* a unique P-solo schedule $S = (q_1, x_1), (q_2, x_2), \ldots, (q_s, x_s)$ of $\mathsf{Cons}_{\mathcal{D}}$ applicable to I^l such that:

$$\forall i \in \{1, 2, \ldots, s\} : x_i \in \{\lambda, d_i\}.$$

Take any $l \in \{0, 1, \ldots, k\}$. For every $P \subseteq \Pi$, all P-solo schedules of $\mathsf{Cons}_{\mathcal{D}}$ induced by I^l and paths in G_p are pieced together in a tree $\Upsilon_p^{P,l}$, called the *simulation tree induced by P, I^l and G_p*, and defined as follows. The set of vertices of $\Upsilon_p^{P,l}$ is the set of *finite* P-solo schedules that are induced by I^l and paths in G_p. The root of $\Upsilon_p^{P,l}$ is the empty schedule $S_\perp$. There is an edge from a vertex S to a vertex S' whenever $S' = S \cdot e$ for some step e; the edge is labeled e.

The construction of $\Upsilon_p^{P,l}$ implies that, for any vertex S of $\Upsilon_p^{P,l}$, there exists a partial run $\langle F, H, I^l, S, T\rangle$ of $\mathsf{Cons}_{\mathcal{D}}$ where F is the current failure pattern and $H \in \mathcal{D}(F)$ is the current failure detector history.

We *tag* every vertex S of $\Upsilon_p^{P,l}$ according to the values p decides in the descendants of S in $\Upsilon_p^{P,l}$: S is assigned a tag v if and only if S has a descendant S' such that p decides v in $S'(I^l)$. The set of all tags of S is called the *valence* of S and denoted *val*(S). If S has only one tag $u \in \{0, 1\}$, then S is called *u-valent.* A 0-valent or 1-valent vertex is called *univalent.* A vertex is called *bivalent* if it has both tags 0 and 1.

Thanks to the reliable communication guarantees provided by registers, for any two correct processes p and q and any time t, there is a time $t' \geq t$ such that $G_p(t) \subseteq G_q(t')$ and, respectively, $\Upsilon_p^{P,l}(t) \subseteq \Upsilon_q^{P,l}(t')$. As a result, and because the DAGs G_p and the simulation trees $\Upsilon_p^{P,l}$ of correct processes p grow monotonically with time, the DAGs tend to the same infinite DAG G, and the simulation trees tend to the same infinite simulation tree $\Upsilon^{P,l}$ [14].

Assume that *correct*$(F) \subseteq P$. By construction, every vertex of $\Upsilon^{P,l}$ has an extension in $\Upsilon^{P,l}$ in which every correct process takes infinitely many steps. By the Termination property of consensus, this extension has a finite prefix S' such that every correct process has decided in $S'(I^l)$. Thus, every vertex S of $\Upsilon^{P,l}$ has a non-empty valence, i.e., S is univalent or bivalent. More generally:

Lemma 6.3 *Let correct*$(F) \subseteq P \subseteq \Pi$, $m \geq 1$, *and* $S_0, S_1, \ldots, S_m$ *be any vertices of* $\Upsilon^{P,l}$. *There exists a finite schedule* S *containing only steps of correct processes such that*

(1) $S_0 \cdot S$ *is a vertex of* $\Upsilon^{P,l}$ *and all correct processes have decided in* $S_0 \cdot S(I^l)$, *and*

(2) for $i \in \{1, 2, \ldots, m\}$, *if* S *is applicable to* $S_i(I^l)$, *then* $S_i \cdot S$ *is a vertex of* $\Upsilon^{P,l}$.

Proof: Let g_i, $i = 0, 1, \ldots, m$, denote any path in G such that S_i is compatible with g_i (by the construction of $\Upsilon^{P,l}$, such paths exist). Since each G_p satisfies properties (1)–(4) of Section 3.1.2, it follows that G contains an infinite path g such that (a) $\forall i \in \{0, 1, \ldots, m\}$, $g_i \cdot g$ is a path in G, (b) the faulty processes do not appear in g, and (c) the correct processes appear infinitely often in g.

Thus, there is a finite schedule S compatible with a prefix of g and applicable to $S_0(I^l)$ such that all correct processes have decided in $S_0 \cdot S(I^l)$. By construction, S contains only steps of correct processes. Recall that, $\forall i \in \{1, 2, \ldots, m\}$, $g_i \cdot g$ is a path in G. Thus, if S is applicable to any $S_i(I^l)$, $i = 1, 2, \ldots, m$, then $S_i \cdot S$ is a vertex of $\Upsilon^{P,l}$. □

The following lemma will facilitate the proof of correctness of our reduction algorithm.

Lemma 6.4 *Let correct*$(F) \subseteq P \subseteq \Pi$. *Let* S_0 *and* S_1 *be any two univalent vertices of* $\Upsilon^{P,l}$ *of opposite valence and* V *be any proper subset of* Π. *If* $S_0(I^l)$ *and* $S_1(I^l)$ *differ only in the states of processes in* V, *then* V *includes at least one correct process.*

Proof: Since $S_0(I^l)$ and $S_1(I^l)$ differ only in the states of processes in V, any $(\Pi - V)$-solo schedule applicable to $S_0(I^l)$ is also applicable to $S_1(I^l)$. By contradiction, assume that V includes only faulty processes. By Lemma 6.3, there is a schedule S containing only steps of correct processes (and thus no steps of processes in V) such that all correct processes have decided in $S_0 \cdot S(I^l)$ and $S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$. Since no process in $\Pi - V$ can distinguish $S_0 \cdot S(I^l)$ and $S_1 \cdot S(I^l)$, the correct processes have decided the same values in these two configurations — a contradiction. □

6.3.3 Decision gadgets

In our reduction algorithm, we exploit the notion of *decision gadgets* [14] by defining forks and hooks similarly to [14], and introducing a new kind of decision gadget which we call a *rake.* Informally, a rake can be used to identify a set of "confused" processes that are not able to distinguish two configurations $S_0(I^l)$ and $S_1(I^l)$ such that S_0 and S_1 are univalent vertices of opposite valence, and these confused processes can only get a decision value from other ("non-confused") processes. Therefore, at least one non-confused process must be correct.

Let $correct(F) \subseteq P \subseteq \Pi$. A *decision gadget* γ is a finite subtree of $\Upsilon^{P,l}$ rooted at $S_\perp$ that includes a vertex $\bar{S}$ (called the *pivot* of the gadget) such that one of the following conditions is satisfied:

(fork) There are two steps e and e' of the same process q, such that $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.

(hook) There is a step e of a process q and a step e' of a process q' $(q \neq q')$, such that:

(i) $\bar{S} \cdot e' \cdot e$ and $\bar{S} \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.

(ii) q and q' do not access the same object of type T in $\bar{S}(I^l)$.

If there is no $x \in \mathcal{R}_\mathcal{D} \cup \{\lambda\}$ such that $\bar{S} \cdot e \cdot (q', x)$ is a vertex of $\Upsilon^{P,l}$, then q' is called *missing* in the hook γ. Clearly, if q' is correct, then it cannot be missing in γ.

(rake) There is a set $U \subseteq P$, $|U| \geq 2$, and an object X of type T such that each $q \in U$ accesses X in $\bar{S}(I^l)$ (U is called the *participating set* of γ). Let E denote the set of *all* vertices of $\Upsilon^{P,l}$ of the form $\bar{S} \cdot S$ where $S = (q_1, \lambda), (q_2, \lambda), \dots, (q_{|U|}, \lambda)$ and $q_1, q_2, \dots, q_{|U|}$ is a permutation of processes in U (E can be empty). $\bar{S}$, U and E satisfy the following conditions:

(i) There do not exist a $(\Pi - U)$-solo schedule S' and a process $q' \in \Pi - U$, such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$ and q' accesses X in $S \cdot S'(I^l)$.

(ii) If $S \in E$, then S is univalent.

(iii) If $|E| = (|U|)!$, i.e., E includes *all* vertices $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_{|U|}, \lambda)$ such that $q_1, q_2, \cdots, q_{|U|}$ is a permutation of processes in U, then there are at least one 0-valent vertex and at least one 1-valent vertex in E.

Note that if $|E| < (|U|)!$, then there is at least one process $q \in U$ such that for some $\{q_1, q_2, \dots, q_s\} \subseteq U - \{q\}$, $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_s, \lambda)$ is a vertex of $\Upsilon^{P,l}$, and $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_s, \lambda) \cdot (q, \lambda)$ is *not* a vertex of $\Upsilon^{P,l}$. We say that such processes are *missing* in the rake. Clearly, every missing process is in *faulty*(F).

Examples of decision gadgets are depicted in Figure 6.3: (a) a fork where $e = (q, x)$ and $e' = (q, x')$, (b) a hook where $e = (q, x)$, $e' = (q', x')$, and q and q' do not access the same object of type T in $\bar{S}(I^l)$; (c) a complete rake with a participating set $U = \{q_1, q_2\}$ and a set of leaves $E = \{\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda), \bar{S} \cdot (q_2, \lambda) \cdot (q_1, \lambda)\}$, where q_1 and q_2 access the same object of type T in $\bar{S}(I^l)$.

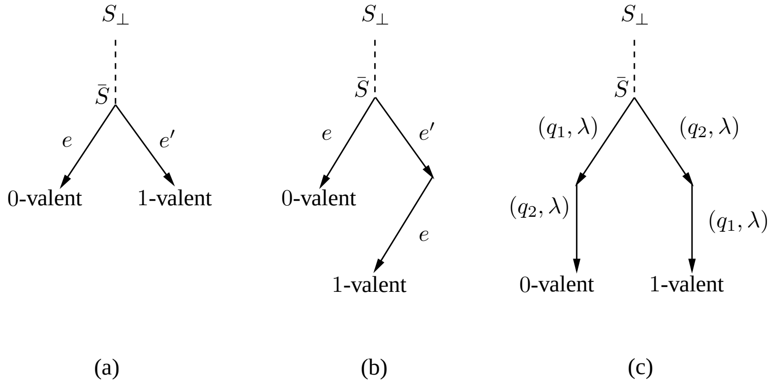

Figure 6.3: A fork, a hook, and a rake

Lemma 6.5 *Let correct(F) $\subseteq P \subseteq \Pi$ and $l \in \{0, 1, \ldots, k\}$. If the root of $\Upsilon^{P,l}$ is bivalent, then $\Upsilon^{P,l}$ contains a decision gadget.*

Proof: Using arguments of Lemma 6.4.1 of [14] (Section 3.1), we can show that there exist a bivalent vertex S^* and a correct process p such that:

(*) For all descendants S' of S^* (including $S' = S^*$) and all $x \in \mathcal{R}_\mathcal{D} \cup \{\lambda\}$ such that $S' \cdot (p, x)$ is a vertex of $\Upsilon^{P,l}$, $S' \cdot (p, x)$ is univalent.

Moreover, one of the following conditions is satisfied:

(1) There are two steps e and e' of p, such that $S^* \cdot e$ and $S^* \cdot e'$ are vertices of $\Upsilon^{P,l}$ of opposite valence.

(2) There is a step e of p and a step e' of a process q such that $S^* \cdot e$ and $S^* \cdot e' \cdot e$ are vertices of $\Upsilon^{P,l}$ of opposite valence.

If (1) is satisfied, then a fork is identified.

Consider case (2). If $p = q$, then by condition (*), $S^* \cdot e'$ is a univalent vertex of $\Upsilon^{P,l}$, and a fork is identified.

Now assume that $p \neq q$. If p and q do not access the same object of type T in $S^*(I^l)$, then we have a hook.

The only case left is when p and q access the same object X of type T in $S^*(I^l)$. The procedure described in Figure 6.4 locates a rake in $\Upsilon^{P,l}$.

We show first that the procedure terminates. Indeed, eventually, either $U = \Pi$, and the algorithm trivially terminates, because there is no $q' \in \Pi - U$, or the algorithm terminates earlier with some $U \subset \Pi$.

Thus, we obtain a set U ($|U| \geq 2$) and a vertex $\bar{S} = S^* \cdot S''$ such that p or q take no steps in S'', S'' applied to $S^*(I^l)$ does not access X (this is ensured by always choosing a shortest schedule in line 10), and every $q' \in U$ accesses X in $\bar{S}(I^l)$. Furthermore:

(i) There do not exist a $(\Pi - U)$-solo schedule S' and a process $q' \in \Pi - U$, such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$ and q' accesses X in $S \cdot S'(I^l)$.

```
$U \leftarrow \{p, q\}$
$\bar{S} \leftarrow S^*$
if $\langle \bar{S} \cdot e \cdot e'$ is vertex of $\Upsilon^{P,l}$ $\rangle$ then
    $E \leftarrow \{\bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$
else
    $E \leftarrow \{\bar{S} \cdot e' \cdot e\}$
while true do
    if $\langle$ there exist a $(\Pi - U)$-solo schedule $S'$ and a process $q' \in \Pi - U$
            such that $\forall S \in \{\bar{S}\} \cup E$, $S \cdot S' \cdot (q', \lambda)$ is a vertex of $\Upsilon^{P,l}$
            and $q'$ accesses $X$ in $S \cdot S'(I^l)$ $\rangle$
    then
            let $S' \cdot (q', \lambda)$ be a shortest such schedule
            $\bar{S} \leftarrow \bar{S} \cdot S'$
            $U \leftarrow U \cup \{q'\}$
            $E \leftarrow$ the set of all vertices $\bar{S} \cdot S$ of $\Upsilon^{P,l}$
                    such that $S = (q_1, \lambda), (q_2, \lambda), \ldots, (q_{|U|}, \lambda)$
                    where $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in $U$
    else exit
```

Figure 6.4: Locating a rake in $\Upsilon^{P,l}$

(ii) If $S \in E$, then S is univalent.

Indeed, take any $S \in E$. By the algorithm in Figure 6.4, $S = S^* \cdot S'$ such that every process in U takes exactly one step in S'. Since $p \in U$, p takes exactly one step in S'. By (*), S is univalent.

(iii) If $|E| = (|U|)!$, i.e., E includes all vertices $\bar{S} \cdot (q_1, \lambda) \cdot (q_2, \lambda) \cdots (q_{|U|}, \lambda)$ such that $q_1, q_2, \ldots, q_{|U|}$ is a permutation of processes in U, then there is at least one 0-valent vertex and at least one 1-valent vertex in E.

Indeed, assume that $|E| = (|U|)!$. By construction, $S^* \cdot S'' \cdot e' \cdot e$, $S^* \cdot S'' \cdot e \cdot e'$, $S^* \cdot e' \cdot e \cdot S''$ and $S^* \cdot e' \cdot e \cdot S''$, where $e = (p, \lambda)$ and $e' = (q, \lambda)$, are vertices of $\Upsilon^{P,l}$. Since S'' applied to $S^*(I^l)$ does not access X, both p and q access X in $S^*(I^l)$, and p or q take no steps in S'', it follows that $S^* \cdot S'' \cdot e' \cdot e(I^l) = S^* \cdot e' \cdot e \cdot S''(I^l)$ and $S^* \cdot S'' \cdot e \cdot e'(I^l) = S^* \cdot e \cdot e' \cdot S''(I^l)$. But $S^* \cdot e \cdot e'$ and $S^* \cdot e' \cdot e$ are univalent vertices of opposite valence. Thus, $S^* \cdot S'' \cdot e \cdot e'$ and $S^* \cdot S'' \cdot e' \cdot e$ are also univalent vertices of opposite valence. Since E includes at least one descendant of $S^* \cdot S'' \cdot e \cdot e'$ and at least one descendant of $S^* \cdot S'' \cdot e' \cdot e$, it follows that there are at least one 0-valent vertex and at least one 1-valent vertex in E.

Hence, a rake with pivot $\bar{S}$ and participating set U is located. □

6.3.4 Critical index

We say that index $l \in \{1, 2, \ldots, k\}$ is *critical in P* if *either* $\Upsilon^{P,l}$ contains a decision gadget, *or* the root of $\Upsilon^{P,l-1}$ is 0-valent, and the root of $\Upsilon^{P,l}$ is 1-valent. In the first case, we say that l is *bivalent critical.* In the second case, we say that l is *univalent critical.*

Lemma 6.6 *Let correct(F) $\subseteq P \subseteq \Pi$. There exists a critical index in P.*

Proof: By Validity of consensus, $\Upsilon^{P,0}$ is 0-valent and $\Upsilon^{P,k}$ is 1-valent. Hence, there exists $l \in \{1, 2, \ldots, k\}$ such that the root of $\Upsilon^{P,l-1}$ is 0-valent and the root of $\Upsilon^{P,l}$ is either 1-valent or bivalent. If the root of $\Upsilon^{P,l}$ is 1-valent, then l is univalent critical. If the root of $\Upsilon^{P,l}$ is bivalent, then, by Lemma 6.5, $\Upsilon^{P,l}$ contains a decision gadget. Thus, l is critical. □

6.3.5 Complete decision gadgets

If a decision gadget γ has no missing processes, we say that γ is *complete.* If γ (a hook or a rake) has a non-empty set of missing processes, we say that γ is *incomplete.*

Lemma 6.7 *Let W be the set of missing processes of an incomplete decision gadget γ. Then $W \subseteq$ faulty(F).*

Proof: Let γ be an incomplete decision gadget of $\Upsilon^{P,l}$ and q be a missing process of γ. By definition, $q \in P$ and there is a vertex S of $\Upsilon^{P,l}$ such that for all $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$, $S \cdot (q, x)$ is not a vertex of $\Upsilon^{P,l}$. This can only happen if q is faulty in F. □

Lemmas 6.5 and 6.7 imply the following:

Corollary 6.8 *Let $C =$ correct(F). If the root of $\Upsilon^{C,l}$ is bivalent, then $\Upsilon^{C,l}$ contains at least one decision gadget, and every decision gadget of $\Upsilon^{C,l}$ is complete.*

6.3.6 Confused processes

The following two lemmas show that a complete hook or a complete rake in $\Upsilon^{P,l}$ can be used to compute a set of one or more "confused" processes that cannot distinguish some two configurations $S_0(I^l)$ and $S_1(I^l)$ such that S_0 and S_1 are univalent vertices of $\Upsilon^{P,l}$ of opposite valence. Later we will use this fact to show how to compute, at the correct processes, the same set of at most n processes including at least one correct process.

Lemma 6.9 *Let correct(F) $\subseteq P \subseteq \Pi$ and γ be a complete hook in $\Upsilon^{P,l}$ defined by a pivot $\bar{S}$, a step e of q, and a step e' of q' ($q \neq q'$). There exists a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S} \cdot e, \bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$ such that:*

(a) S_0 and S_1 are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, and

(b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p.

Proof: By the definition of γ, $\bar{S} \cdot e$ and $\bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, q and q' do not access the same object of type T, and there is a vertex $\bar{S} \cdot e \cdot (q', x)$ in $\Upsilon^{P,l}$ for some $x \in \mathcal{R}_{\mathcal{D}} \cup \{\lambda\}$.

Assume that $e' = (q', \lambda)$, and q and q' do not access the same register in $\bar{S}(I^l)$. Thus, $\bar{S} \cdot e \cdot e'$ is a vertex of $\Upsilon^{P,l}$ such that $\bar{S} \cdot e \cdot e'(I^l) = \bar{S} \cdot e' \cdot e(I^l)$. But $\bar{S} \cdot e$

and $\bar{S} \cdot e' \cdot e$ have opposite valences — a contradiction. Thus either (1) q' queries $\mathcal{D}$ in $\bar{S}(I^l)$, or (2) q and q' access the same register in $\bar{S}(I^l)$.

(1) If q' queries $\mathcal{D}$ in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q'.

(2) Assume now that q and q' access the same register X in $\bar{S}(I^l)$. Thus, $e = (q, \lambda)$, $e' = (q', \lambda)$, and $\bar{S} \cdot e \cdot e'$ is a univalent vertex of $\Upsilon^{P,l}$. Furthermore:

- If q writes in X in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q'.
- If q reads X in $\bar{S}(I^l)$, then $S_0 = \bar{S} \cdot e \cdot e'$ and $S_1 = \bar{S} \cdot e' \cdot e$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence such that $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of q.

In each case, we obtain a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S} \cdot e, \bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$ such that (a) S_0 and S_1 are univalent vertices of $\Upsilon^{P,l}$ of opposite valence, and (b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p. □

The following lemma uses the assumptions that type T is deterministic and *cons*$(T) \leq n$.

Lemma 6.10 *Let correct$(F) \subseteq P \subseteq \Pi$ and γ be a complete rake in $\Upsilon^{P,l}$ with a pivot $\bar{S}$ and a participating set U such that $|U| = n+1$. Let E be the set of leaves of γ and X be the object of type T accessed by processes in U in $\bar{S}(I^l)$. There exist a process $p \in U$ and two univalent vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E such that*

(a) val$(\bar{S} \cdot S_0) \neq$ val$(\bar{S} \cdot S_1)$, and

(b) $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ differ only in the states of processes in $U - \{p\}$ and object X.

Proof: Let γ be a complete rake with a pivot $\bar{S}$ and a participating set U such that $|U| = n + 1$. Let X be the object of type T such that each process $q \in U$ accesses X in $\bar{S}(I^l)$. Let σ_X be the state of X in $\bar{S}(I^l)$.

Construct a graph $\mathcal{K}$ as follows. The set of vertices of $\mathcal{K}$ is E, the set of leaves of γ. Two vertices $\bar{S} \cdot S$ and $\bar{S} \cdot S'$ of $\mathcal{K}$ are connected with an edge if at least one process in U has the same state in $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$. Now we color each vertex $\bar{S} \cdot S$ of $\mathcal{K}$ with *val*$(\bar{S} \cdot S)$.

Claim 6.10.1 *Vertices of $\mathcal{K}$ are colored* 0 *or* 1, *and $\mathcal{K}$ has at least one vertex of color* 0 *and at least one vertex of color* 1.

Proof of Claim 6.10.1. Immediate from the definition of $\mathcal{K}$. □

Now we show that $\mathcal{K}$ is connected. By contradiction, assume that $\mathcal{K}$ consists of two or more connected components. Let $\mathcal{K}_0$ be any connected component of $\mathcal{K}$

and $\mathcal{K}_1 = \mathcal{K} - \mathcal{K}_0$. Note that for any two vertices $S \in \mathcal{K}_0$ and $S' \in \mathcal{K}_1$, no process has the same state in $S(I^l)$ and $S'(I^l)$. We establish a contradiction by showing that consensus can be solved among $n+1$ processes using registers and objects of type T.

Claim 6.10.2 *There is an algorithm that solves* weak consensus *among $n+1$ processes using registers and one object of type T.*

Proof of Claim 6.10.2. Let X, an object of type T, be initialized to σ_X. Every process $p \in U$ executes one step of $\mathsf{Cons}_{\mathcal{D}}$ determined by p's state in $\bar{S}(I^l)$ (by the definition of γ, in this step, p accesses X). If its resulting state belongs to a configuration $S(I^l)$ such that $S \in \mathcal{K}_0$, then p decides 0. Otherwise, p decides 1.

Termination is trivially ensured. Since the process state after taking one step of σ_X unambiguously identifies to which component the resulting configuration belongs, Agreement is satisfied. Non-Triviality follows from the fact that $\mathcal{K}_0$ and $\mathcal{K}_1$ are non-empty. □

By Claim 6.10.2 and Lemma 6.2, since T is deterministic, we can solve consensus among $n+1$ processes using registers and one object of type T — a contradiction of the assumption that *cons*$(T) \leq n$. Thus, $\mathcal{K}$ is connected.

By Claim 6.10.1 and the fact that $\mathcal{K}$ is connected, $\mathcal{K}$ has at least two vertices of different colors, connected by an edge. From the construction of $\mathcal{K}$, it follows that there exist a process $p \in U$ and two univalent vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ of opposite valence in E such that $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ differ only in the states of processes in $U - \{p\}$ and object X. □

6.3.7 Deciding sets

Instead of the notion of a *deciding process* used in [14], we introduce the notion of a *deciding set* $V \subset \Pi$. The deciding set V of a *complete* decision gadget γ is defined as follows:

(1) Let γ be a fork defined by pivot $\bar{S}$ and steps e and e' of the same process q, such that $\bar{S} \cdot e$ and $\bar{S} \cdot e'$ are univalent vertices of $\Upsilon^{P,l}$ of opposite valence.

Then $V = \{q\}$.

(2) Let γ be a complete hook defined by a pivot $\bar{S}$, a step e of q, and a step e' of q' $(q \neq q')$.

By Lemma 6.9, there exists a process $p \in \{q, q'\}$ and two vertices S_0 and S_1 in $\{\bar{S} \cdot e, \bar{S} \cdot e' \cdot e, \bar{S} \cdot e \cdot e'\}$ such that (a) S_0 and S_1 are univalent vertices of opposite valence, and (b) $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p. Then we define $V = \{p'\}$ where p' is the smallest such process.

(3) Let γ be a complete rake defined by a pivot $\bar{S}$, a participating set U, and a set of leaves E.

- If $|U| \leq n$, then we define $V = U$.

- If $|U| = n + 1$, then by Lemma 6.10 there is a process $p \in U$ such that, for some $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E, *val*$(\bar{S} \cdot S_0) \neq$ *val*$(\bar{S} \cdot S_1)$, and $\bar{S} \cdot S(I^l)$ and $\bar{S} \cdot S'(I^l)$ differ only in the states of processes in $U - \{p\}$ and object X. Then we define $V = U - \{p'\}$ where p' is the smallest such process.

By definition, in each case, V is a set of at most n processes. The following lemma uses the assumption that type T is *one-shot*.

Lemma 6.11 *The deciding set of a complete decision gadget contains at least one correct process.*

Proof: There are three cases to consider:

(1) Let γ be a fork with leaves $\bar{S} \cdot (p, x)$ and $\bar{S} \cdot (p, x)$, The deciding set of γ is $\{p\}$. Since we consider here deterministic algorithms and objects, it follows that p queries $\mathcal{D}$ in $\bar{S}(I^l)$. Thus, $S_0(I^l)$ and $S_1(I^l)$ differ only in the state of p. By Lemma 6.4, p is correct.

(2) Let γ be a hook with a deciding set $V = \{p\}$. By Lemma 6.4, p is correct.

(3) Let γ be a complete rake defined by a pivot $\bar{S}$, a participating set U, and a set of leaves E. Let X be the object of type T that processes in U access in $\bar{S}(I^l)$. The following cases are possible:

(3a) $|U| \leq n$.

There exist two vertices $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$ in E such that *val*$(\bar{S} \cdot S_0) = 0$ and *val*$(\bar{S} \cdot S_1) = 1$. Since only processes in U take steps in S_0 and S_1, and both schedules applied to $\bar{S}(I^l)$ access only object X, the configurations $\bar{S}(I^l)$, $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ differ only in the states of processes in U and object X. Assume, by contradiction, that all processes in the deciding set $V = U$ are faulty.

By Lemma 6.3, there is a schedule S containing only steps of correct processes (and thus no steps of processes in U) such that all correct processes have decided in $\bar{S} \cdot S(I^l)$ and, for every $S' \in E$, if S is applicable to $S'(I^l)$, then $S' \cdot S$ is a vertex of $\Upsilon^{P,l}$.

Suppose that S applied to $\bar{S}(I^l)$ accesses X, i.e., S has a prefix $S'' \cdot (q, \lambda)$ such that S'' applied to $\bar{S}(I^l)$ does not access X and q accesses X in $\bar{S} \cdot S''(I^l)$.

Take any $S' \in E$. Since $S'(I^l)$ and $\bar{S}(I^l)$ differ only in the states of processes in U and the state of X, S'' includes no steps of processes in U, and S'', applied to $\bar{S}(I^l)$, does not access X, it follows that $S'' \cdot (q, \lambda)$ is applicable to $S'(I^l)$, Thus, for all $S' \in E$, $S' \cdot S'' \cdot (q, \lambda)$ is a vertex of $\Upsilon^{P,l}$ — a contradiction with the definition of γ.

Thus, S applied to $\bar{S}(I^l)$ does not access X. Hence, S is also applicable to $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$. Thus, $\bar{S} \cdot S_0 \cdot S$ and $\bar{S} \cdot S_1 \cdot S$ are vertices of $\Upsilon^{P,l}$.

But no process in $\Pi - U$ can distinguish $\bar{S} \cdot S(I^l)$, $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$. Thus, the correct processes have decided the same values in these configurations — a contradiction.

(3b) $|U| \geq n+1$, i.e., $U = \Pi$. Let $V = U - \{p\}$ be the deciding set of γ, i.e., for some $\bar{S} \cdot S_0$ and $\bar{S} \cdot S_1$, the vertices of $\Upsilon^{P,l}$ of opposite valence, $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$ differ only in the states of processes in V and object X. Assume, by contradiction, that all processes in V are faulty (i.e., since $k = n+1$, the only correct process is p).

By Lemma 6.3, there is a schedule S containing only steps of correct processes (i.e., only steps of p) such that all correct processes have decided in $\bar{S} \cdot S_0 \cdot S(I^l)$, and if S is applicable to $\bar{S} \cdot S_1(I^l)$, then $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$.

Note that, since X is an object of a one-shot type, and p has already accessed X at least once in $\bar{S} \cdot S_0(I^l)$, every subsequent operation of p on object X returns $\perp$. Since the states of p and all objects except of X are the same in $\bar{S} \cdot S_0(I^l)$ and $\bar{S} \cdot S_1(I^l)$, and p has already accessed X at least once in $\bar{S} \cdot S_1(I^l)$, S is also applicable to $\bar{S} \cdot S_1(I^l)$ and p cannot distinguish $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$. Thus, $\bar{S} \cdot S_1 \cdot S$ is a vertex of $\Upsilon^{P,l}$, and p has decided the same value in $\bar{S} \cdot S_0 \cdot S(I^l)$ and $\bar{S} \cdot S_1 \cdot S(I^l)$ — a contradiction.

In each case, the deciding set V contains at least one correct process. □

6.3.8 Reduction algorithm

Theorem 6.12 *Let T be any one-shot deterministic type, such that cons$(T) \leq n$ and $n \geq 1$. If a failure detector $\mathcal{D}$ solves consensus in a system of $k = n+1$ processes using only registers and objects of type T, then $\Omega_n \preceq \mathcal{D}$.*

Proof: Let F be any failure pattern and $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus in a system of $k = n+1$ processes using $\mathcal{D}$, registers, and objects of type T.

The communication task presented in Figure 6.2 and the computation task presented in Figure 6.5 constitute the reduction algorithm $T_{\mathcal{D} \to \Omega_n}$. The current estimate of Ω_n at process p is stored in variable Ω_n-*output*$_p$.

In the communication task described in Figure 6.2, every process p maintains an ever-growing DAG G_p.

In the computation task described in Figure 6.5, for each $Q \subseteq \Pi$ and each $l \in \{0, 1, \ldots, k\}$, process p constructs a finite simulation tree $\Upsilon_p^{Q,l}$ induced by Q, I^l and G_p and tags each vertex S of $\Upsilon_p^{Q,l}$ with a set of decisions p takes in all $S'(I^l)$ such that S' is a descendant of S in $\Upsilon_p^{Q,l}$.

We say that index $l \in \{1, 2, \ldots, k\}$ is *p-critical in P* if *either* $\Upsilon_p^{P,l}$ contains a decision gadget *or* the root of $\Upsilon_p^{P,l-1}$ is 0-valent, and the root of $\Upsilon_p^{P,l}$ is 1-valent. In the first case, we say that l is *bivalent p-critical.* In the second case, we say that l is *univalent p-critical.* Decision gadgets, missing processes, and deciding sets are defined for each finite simulation tree $\Upsilon_p^{P,l}$ in the same way as for the "limit" simulation tree $\Upsilon^{P,l}$.

In every round of the task, trees $\Upsilon_p^{P,l}$ are analyzed for several values of $P \subseteq \Pi$. Initially $P = \Pi$. Let P have a p-critical index and let l be the smallest p-critical

```
Initially:
    $\Omega_n$-$output_p \leftarrow \{p\}$

while true do
    for all $Q \subseteq \Pi$ and $l \in \{0, 1, \ldots, k\}$ do
        $\Upsilon_p^{Q,l} \leftarrow$ simulation tree induced by $Q$, $I^l$ and $G_p$
    $V \leftarrow \emptyset$
    $P \leftarrow \Pi$
    repeat
        if $P$ has no $p$-critical index then
            $V \leftarrow \{p\}$
        else
            let $l$ be the smallest $p$-critical index of $P$
            if $l$ is univalent $p$-critical then
                $V \leftarrow \{p_l\}$
            else
                $\gamma \leftarrow$ the smallest decision gadget in $\Upsilon_p^{P,l}$
                if $\gamma$ is complete  then
                    $V \leftarrow$ the deciding set of $\gamma$
                else
                    let $W$ be the set of missing processes in $\gamma$
                    $P \leftarrow P - W$
    until  $V \neq \emptyset$ or $P = \emptyset$
    if $P = \emptyset$ then $V \leftarrow \{p\}$
    $\Omega_n$-$output_p \leftarrow V$
```

Figure 6.5: Extracting Ω_n: code for each process p

index in P. If l is univalent, then p outputs $\{p_l\}$ (line 12). If l is bivalent, and the smallest decision gadget in $\Upsilon_p^{P,l}$, denoted γ, is complete, then p outputs the deciding set of γ (line 16). If l is bivalent, and γ is incomplete, then p removes missing (in γ) processes from P and proceeds to the next iteration (line 19).

Note that the "repeat-until" cycle in lines 6–20 is non-blocking. Indeed, eventually, p either sets V to a non-empty value (in lines 8, 12 or 16), or sets P to $\emptyset$ in line 19. In both cases, p exits the "repeat-until" cycle.

Recall that finite simulation trees $\Upsilon_p^{P,l}$ at all correct processes p tend to the same *infinite* simulation tree $\Upsilon^{P,l}$. Let F be the current failure pattern.

Claim 6.12.1 *There exist $P^*, V^* \subseteq \Pi$, correct$(F) \subseteq P^*$, such that there is a time after which every correct process p always reaches line 21 with $P = P^*$ and $V = V^*$.*

Proof of Claim 6.12.1. By Lemma 6.6, every $P \subseteq \Pi$ such that *correct*$(F) \subseteq P$ has a critical index. Thus, there is a time after which all correct processes p locate the same P such that P has a critical index l, and l is also the smallest p-critical index in P. Moreover, if l is bivalent critical, then the correct processes eventually locate the same smallest (complete or incomplete) decision gadget in $\Upsilon^{P,l}$.

By Lemma 6.7, there is a time after which, whenever a correct process p reaches line 19, $W \subseteq$ *faulty*(F). Thus, there is a time after which one of the following cases always holds:

(a) p exits the "repeat-until" cycle in line 12 after having located a univalent p-critical index in some P such that *correct*$(F) \subseteq P$.

(b) p exits the "repeat-until" cycle in line 16 after having located a complete decision gadget in univalent p-critical index in $\Upsilon^{P,l}$ where $P \subseteq \Pi$ and *correct*$(F) \subseteq P$.

(c) p reaches line 14 with $P =$ *correct*(F).

In case (c), by Corollary 6.8, there is a time after which the smallest decision gadget in $\Upsilon^{P,l}$ is complete and p exits the "repeat-until" cycle in line 16. In all cases, there exist $P^*, V^* \subseteq \Pi$ such that *correct*$(F) \subseteq P^*$, and a time after which every correct process always reaches line 21 with $P = P^*$ and $V = V^*$. □

Let l be the smallest critical index in P^*. There is a time after which, for all correct processes p, l is also the smallest p-critical in P^*. By the algorithm, only the following cases are possible:

(1) l is univalent critical, i.e., the root of $\Upsilon^{P^*,l-1}$ is 0-valent and the root of $\Upsilon^{P^*,l}$ is 1-valent. In this case, eventually, every correct process p permanently outputs $V^* = \{p_l\}$. I^{l-1} and I^l differ only in the state of process p_l. By Lemma 6.3, p_l is correct.

(2) l is bivalent critical, i.e., $\Upsilon^{P^*,l}$ contains a decision gadget. Moreover, by the algorithm, the smallest decision gadget in $\Upsilon^{P^*,l}$ is complete. In this

case, eventually, every correct process p permanently outputs the deciding set V^* (of size at most n) of the complete decision gadget. By Lemma 6.11, the deciding set of γ includes at least one correct process.

In both cases, eventually, the correct processes agree on a set of at most n processes that include at least one correct process. Thus, the reduction algorithm described in Figures 6.2 and 6.5 emulates the output of Ω_n. $\square$

Theorem 6.12 and the algorithm of [61] imply the following:

Theorem 6.13 *Let T be any one-shot deterministic type such that cons$(T) = n$ and $n \geq 1$. Then Ω_n is the weakest failure detector to solve consensus in a system of $k = n + 1$ processes using registers and objects of type T.*

Note that our reduction algorithm (Figure 6.5) is allowed to eventually output a set of two or more processes only when a rake is located. In all other cases, it eventually outputs a single correct process, which complies with the specification of Ω. But a rake can be located only if the underlying consensus algorithm $\mathsf{Cons}_{\mathcal{D}}$ employs objects of type T. Thus, as a corollary, assuming that only registers are available, we obtain the following result, stated in [53].

Corollary 6.14 *For every $k \geq 2$, Ω is the weakest failure detector to solve consensus among k processes using only registers.*

6.4 Boosting consensus power to any level $k > n$

The results of Section 6.3 can be generalized as follows.

Theorem 6.15 *Let T be any m-*ported *one-shot deterministic type, such that $m \leq n+1$ and cons$(T) \leq n$. If a failure detector $\mathcal{D}$ solves consensus in a system of k ($k > n$) processes using only registers and objects of type T, then $\Omega_n \preceq \mathcal{D}$.*

Proof: Let F be any failure pattern and $\mathsf{Cons}_{\mathcal{D}}$ be any algorithm that solves consensus in a system of k processes using $\mathcal{D}$, registers, and objects of type T. The reduction algorithm $T_{\mathcal{D} \to \Omega_n}$ is similar to the algorithm described in Figure 6.5. The definitions of decision gadgets and deciding sets are equivalent to the ones of Section 6.3. Indeed, since type T is at most $(n+1)$-ported, the participating set U of any rake γ can include at most $n+1$ processes. Respectively, if $|U| \leq n$, then the deciding set V of γ is U. If $|U| = n+1$, then, by Lemma 6.10 there is at least one "confused" process p, and the deciding set V is defined as $U - \{p\}$. Clearly, $|V| = n$. One can easily see that V must be alive. Otherwise, there is a $(\Pi - V)$-solo schedule, in which every correct process takes an infinite number of steps such that no process can distinguish a 0-valent vertex from a 1-valent one. The deciding sets of forks and hooks do not depend on the system size. $\square$

Theorem 6.15 and [61] imply the following:

Theorem 6.16 *Let T be any m-ported one-shot deterministic type such that $m \leq n+1$ and cons(T) $= n$. Then Ω_n is the weakest failure detector to solve consensus in a system of k ($k > n$) processes using registers and objects of type T.*

6.5 Boosting object resilience

So far we assumed the one-to-one static binding scheme [13], according to which, in every run, an object port can be accessed by at most one process, and a process is allowed to access at most one port of every object. Furthermore, we considered systems in which processes communicate through *wait-free* linearizable implementations of shared objects. Every process can complete every operation on an object in a finite number of its own steps, regardless of the behavior of other processes.

In contrast, in this section, we assume the most permissive *softwired* binding scheme [13]. In the softwired binding scheme, a process is allowed to apply operations to any number of ports on an object, and the process "owns" a port only for a duration of its operation. For consensus objects, the distinction between the softwired and one-to-one static binding schemes is however negligible [10, 13]:

Lemma 6.17 *For all $m, k \in \mathbb{N}$, there is an implementation of a wait-free softwired m-process consensus object shared by k processes from a bounded number of wait-free m-process consensus base objects such that the binding with every base object is one-to-one static.*

Moreover, in this section, we consider *t-resilient* linearizable implementations of object types (we will simply call these *t-resilient objects*). The liveness properties of a t-resilient object only guarantee that a correct process completes its operation on the object port, as long as no more than t processes that access other ports of the object crash in the middle of their operations (this corresponds to *strongly t-resilient* implementations [13]).

Assume that k processes communicate through registers and t-resilient objects of *any* object types (not necessarily one-shot deterministic types with a known bound on the number of ports).

In this section, we show that Ω_{t+1} is the weakest failure detector to solve consensus in this system.

6.5.1 Boosting resilience of atomic objects is impossible

We show first that it is impossible to solve $(t+1)$-resilient consensus among $n > t+1$ processes using registers and t-resilient atomic objects (without failure detectors).

The following two lemmas correspond to the "necessity" part and the "sufficiency" part of Theorem 4.1 in [12], respectively.

Lemma 6.18 *Let t and n be integers, $0 \leq t$, $1 \leq n$. Then there exists a t-resilient n-process implementation of consensus from wait-free $(t+1)$-process consensus* *objects and wait-free registers.*[3]

Lemma 6.19 *Let t and n be integers, $2 \leq t < n$. Then there exists a wait-free implementation of $(t+1)$-process consensus from t-resilient n-process consensus objects and wait-free registers.*

The following result follows easily from Herlihy's universal construction [41]:

Lemma 6.20 *Let t and n be integers, $0 \leq t$, $1 \leq n$. Let T be any n-ported type. Then there exists a t-resilient implementation of an atomic object of type $\mathcal{T}$ from t-resilient n-process consensus objects and wait-free registers.*

The following result is shown in [41, 46]:

Lemma 6.21 *Let n be an integer, $n \geq 0$. There does not exist a wait-free implementation of $(n+1)$-process consensus from wait-free n-process consensus objects and wait-free registers.*

Theorem 6.22 *Let t and n be integers, $0 \leq t < n-1$. There does not exist a $(t+1)$-resilient n-process implementation of consensus from t-resilient atomic objects and wait-free registers.*

Proof: By contradiction, assume that there exists a $(t+1)$-resilient n-process implementation of consensus from t-resilient atomic objects and wait-free registers. We consider two cases:

First suppose that $t = 0$, so $n \geq 2$. Thus, we have a 1-resilient n-process implementation of consensus using 0-resilient atomic objects and wait-free registers. By Lemma 6.20, each 0-resilient atomic object used in this implementation can itself be implemented from 0-resilient consensus objects and wait-free registers. By substituting these implementations for the objects, we obtain a 1-resilient n-process implementation of consensus using 0-resilient consensus objects and wait-free registers.

A 0-resilient consensus object shared by any number of processes can be easily implemented from two wait-free registers as follows. Every process participating in the consensus algorithm writes its input value in a dedicated "proposal" register X (initialized to $\perp$). Then the process keeps reading a dedicated "decision" register D (initialized to $\perp$) until a non-$\perp$ value is read, in which case the process decides on this value. In parallel, a dedicated process p keeps reading X. As soon as p reads a non-$\perp$ value v in X, p writes v in D.

Substituting once more, we obtain a 1-resilient n-process implementation of consensus using only wait-free registers. But this contradicts the impossibility result of [56].

Now suppose that $t \geq 1$. By Lemma 6.20, each t-resilient atomic object used in this implementation can itself be implemented from t-resilient n-process consensus objects and wait-free registers. By substituting, we obtain a $(t+1)$-resilient n-process implementation of consensus from t-resilient n-process consensus objects and wait-free registers. By Lemma 6.18, each t-resilient n-process

[3] Theorem 4.1 in [12] assumes $2 \leq t$. However, the necessity part of the theorem requires only $0 \leq t$.

consensus object used in this implementation can be implemented from wait-free $(t+1)$-process consensus objects and wait-free registers. By substituting again, we obtain a $(t+1)$-resilient n-process implementation of consensus from wait-free $(t+1)$-process consensus objects and wait-free registers. Now by Lemma 6.19 (using the fact that $2 \leq t+1 < n$), a wait-free $(t+2)$-process consensus object can be implemented from $(t+1)$-resilient n-process consensus objects and wait-free registers. By substituting, we obtain an implementation of a wait-free $(t+2)$-process consensus object from wait-free $(t+1)$-process consensus objects and wait-free registers. But this contradicts Lemma 6.21. □

6.5.2 Boosting resilience of atomic objects with failure detectors

Thus, roughly speaking, it is not possible to obtain a more fault-tolerant system solving consensus by combining less fault-tolerant atomic objects. Not surprisingly, this impossibility can be circumvented by augmenting the system with a failure detector abstraction. Interestingly, our result on boosting the consensus power of one-shot deterministic types (Theorem 6.16) implies the following:

Theorem 6.23 *Let t and k be integers, $k > t \geq 2$. Let T be any type such that registers and t-resilient objects of type T implement t-resilient k-process consensus. Then Ω_{t+1} is the weakest failure detector to solve consensus among k processes using wait-free registers and t-resilient objects of type T.*

Proof: By Lemma 6.19, a wait-free $(t+1)$-process consensus object can be implemented from wait-free registers and t-resilient objects of type T. The algorithm of [61] implements wait-free consensus among k processes using registers, wait-free $(t+1)$-process consensus objects and Ω_{t+1}. This gives the sufficient part of the theorem.

Assume now that a failure detector $\mathcal{D}$ solves consensus among k processes using registers and t-resilient objects of type T. By Lemmas 6.18, 6.19, 6.20 and 6.17, any t-resilient object can be implemented from wait-free registers and $(t+1)$-process consensus objects so that, in the implementation, the binding with each base object is one-to-one static. By recalling Lemma 6.21, we observe that $(t+1)$-process consensus is a $(t+1)$-ported one-shot deterministic type of consensus power $t+1$.

Thus, $\mathcal{D}$ solves consensus using registers and objects of $(t+1)$-ported one-shot deterministic type of consensus power $t+1$ using the one-to-one static binding scheme. By Theorem 6.15, $\Omega_{t+1} \preceq \mathcal{D}$. This gives the necessary part of the theorem. □

6.6 Open questions

Neiger's original conjecture that Ω_n is the weakest failure detector for boosting *any* type T (not necessarily one-shot deterministic type with $n+1$ or less ports) of consensus power n [61] remains unproved. We proved it only for $(n+1)$-ported one-shot deterministic types.

Note that in the proof of correctness of our reduction algorithm $T_{\mathcal{D}\rightarrow\Omega_n}$ (Section 6.3), the assumption that type T is one-shot deterministic is used only in Lemmas 6.10 and 6.11.

To prove Lemma 6.10, we assume the opposite, and we establish a contradiction by showing that T can solve *weak* consensus among $n+1$ processes. For deterministic types, we show that T also solves consensus among $n+1$ processes. For non-deterministic types, this is not the case. In [36], we present a non-deterministic type that solves weak consensus among any number of processes, but cannot solve 2-process consensus. Furthermore, certain non-deterministic types exhibit unusual behavior with respect to the *robustness* question [11, 65, 54].

To prove Lemma 6.11, we use the fact that if the decision value is "locked" in the state of a *one-shot* object X, then the only way for a fixed set W of one or more "confused" processes to fetch the value is to communicate with at most n other processes. This implies that at least one process in $\Pi - W$ must be correct. If X is multi-shot, this is not necessarily true. However, by Lemma 6.10, if processes try to fetch the decision *only* by invoking a bounded number of operations on X, then at least one of them will be confused. But it is still not clear whether it is possible to extract Ω_n using this fact.

Finally, in the case $k > n+1$, it is very appealing to get rid of the assumption that type T is at most $(n+1)$-ported. This assumption was used to prove Lemmas 6.5 and 6.11, and it is not very clear whether these two results hold without the bound on the number of ports of type T.

6.7 Related work

The notion of consensus power was introduced by Herlihy [41] and then refined by Jayanti [45].

Chandra, Hadzilacos and Toueg [14] showed that Ω is the weakest failure detector to solve consensus in asynchronous message-passing systems with a majority of correct processes. Lo and Hadzilacos [53] showed that Ω can be used to solve consensus with registers and stated (without proof) that any failure detector that can be used to solve consensus with registers can be transformed to Ω.

Neiger [61] introduced the hierarchy of failure detectors Ω_n and showed that objects of consensus power n can solve consensus among any number of processes using Ω_n.

The relationship between consensus and weak consensus with respect to deterministic object types was established in [36]. An indirect proof that it is impossible to boost the resilience of atomic objects without using failure detectors, based on the results of [41, 46, 12], appeared in [34]. A direct self-contained proof of this result appeared in [6], and then it was extended to more general classes of *distributed services* in [7].

Chapter 7

Concluding Remarks

We conclude the thesis by discussing a generic framework that describes systems composed of *distributed services* (e.g., failure detectors, reliable channels, or atomic objects) and proposing some future research directions.

7.1 Failure detectors as distributed services

We discuss here an alternative approach to describing failure detectors. This approach makes it possible to put failure detectors on the same ground as other abstractions in distributed computing. A distributed system is seen here as a collection of processes that communicate through *distributed services.* A distributed service, and, in particular, a failure detector, is defined through a predicate on the traces of the corresponding I/O automaton [57, chapter 8]. Unlike in [14], we avoid using the notion of time by explicitly modeling process failures as *actions* in a computation.

We introduce the notion of a *generic* distributed service which can be used to model the definitions of atomic objects [43, 41], reliable channels [28], and failure detectors [15, 14]. Failure detectors defined in this manner seem to encompass all failure detectors defined in the formalism of Chandra, Hadzilacos and Toueg [14], except that they are not allowed to produce output based on future events. In other words, the failure detectors defined in this chapter are *realistic* [19]. This limitation however does not seem to impact most of important theoretical results in the "failure detector" domain.

Our resulting framework describes systems composed of a collection of distributed services. A version of this framework was used in [7] to prove that boosting *consensus resilience* of generic distributed services is impossible.

7.1.1 I/O automata

An I/O automaton [57, chapter 8] is a special kind of state machine in which the transitions are associated with named *actions.* The actions can be *input*, *output*, or *internal.* Inputs and outputs represent the *external interface* of the automaton used for communication with the automaton's environment. The internal actions represent the internal transitions visible only to the automaton itself. It is assumed that the input actions are out of the automaton's control, unlike the output and internal actions.

A *signature* S is a tuple $\langle \mathit{in}(S), \mathit{out}(S), \mathit{int}(S)\rangle$ consisting of the input actions, $\mathit{in}(S)$, the output actions, $\mathit{out}(S)$, and internal actions, $\mathit{int}(S)$. We define the *locally controlled* actions, $\mathit{loc}(S)$, to be $\mathit{out}(S) \cup \mathit{int}(S)$, the *external* actions, $\mathit{ext}(S)$, to be $\mathit{in}(S) \cup \mathit{out}(S)$, and $\mathit{acts}(S)$ to be all actions of S.

Formally, an *I/O automaton* (from now on simply an *automaton*) A is defined by:

- $\mathit{sig}(A)$, a signature;
- $\mathit{states}(A)$, a (possibly infinite) set of *states*;
- $\mathit{start}(A)$, a non-empty subset of $\mathit{states}(A)$, called the *initial states*;
- $\mathit{trans}(A)$, a *state-transition relation*, where

$$\mathit{trans}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(\mathit{sig}(A)) \times \mathit{states}(A);$$

- $\mathit{tasks}(A)$, a countable partition of actions in $\mathit{loc}(\mathit{sig}(A))$.

An element (s, π, s') of $\mathit{trans}(A)$ is called a *transition* of A. The transition (s, π, s') is called *input transition*, *output transition*, or *internal transition*, if π is, respectively, an input action, output action, or internal action.

If for some state s and action π, A has a transition (s, π, s'), then we say that π is *enabled* in s. By definition, every automaton is *input-enabled*: every input action is enabled in every state. A task e is said to be *enabled* in a state s if some action of e is enabled in s.

An *execution fragment* is either a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_r, s_r$, or an infinite sequence $s_0, \pi_1, s_1, \ldots, \pi_r, s_r, \ldots$, of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. An execution fragment that begins with an initial state is called an *execution*. A *trace* of A is a sequence of external actions of A obtained by removing the states and internal actions from an execution of A. If α and α' are execution fragments of A such that α' starts in the last state of α, then the concatenation $\alpha \cdot \alpha'$ is defined, and is called an *extension* of α.

We model a distributed system as a *parallel composition* of automata representing individual system components. The composition of automata identifies actions with the same name in the component automata. When any component automaton performs a step involving π, so do all component automata that have π in their signatures. We assume that only *compatible* automata can be composed, and we recall the notion of compatibility below. Informally, a countable collection $\{A_i\}_{i \in I}$ of automata is compatible if, for all $i, j \in I$, $i \neq j$, (1) the internal actions of A_i are not observable by A_j, i.e., $\mathit{int}(\mathit{sig}(A_i)) \cap \mathit{acts}(\mathit{sig}(A_j)) = \emptyset$, and (2) the sets of output actions of A_i and A_j are disjoint, i.e. $\mathit{out}(\mathit{sig}(A_i)) \cap \mathit{out}(\mathit{sig}(A_j)) = \emptyset$. In a composition of compatible automata $\{A_i\}_{i \in I}$, outputs of all the components become outputs of the composition, internal actions of the components become internal actions, and actions that are inputs of some components but outputs of none become inputs of the composition. The states and initial states of the composition are vectors of states and initial states of the component automata.

The transitions of the composition are obtained by allowing the components that have a particular action π in their signature to execute simultaneously a step involving π, while all other components do nothing. The task partition of the composition is a union of the components' task partitions. Further, we "hide" actions that are used for communication between components, by reclassifying them as internal actions. This prevents them from being included in traces.

An execution α of A is said to be *fair* if for each task e of A: (1) if α is finite, then e is not enabled in the final state of α, and (2) if α is infinite, then α contains either infinitely many actions of e, or infinitely many occurrences of states in which e is not enabled. A trace of a fair execution is called a *fair trace*. Her we consider only fair executions.

7.1.2 Distributed services

We define a special kind of I/O automata, the *canonical distributed service of type $\mathcal{U}$ for endpoint set J and index k*, where

- service type $\mathcal{U}$ is a tuple $\langle V, V_0, invs, resps, glob, \delta_1, \delta_2\rangle$,
- J is a finite set of *endpoints* at which invocations and responses may occur,
- k is a unique index (name) for the service.

The parameter J specifies the set of processes connected to the service. A process $i \in J$ can issue any allowable invocation on the service and (potentially) receive any allowable response. We allow concurrent (overlapping) operations, at the same or at different endpoints. The service preserves the order among concurrent invocations at the same endpoint i by keeping the invocations and responses in internal FIFO buffers.

V is a non-empty set of *values* that can be stored in a state component *val*, $V_0 \subseteq V$ is a non-empty set of *initial values*, *invs* is a set of *invocations*, *resps* is a set of *responses*, *glob* is a set of *global tasks*, and δ_1, δ_2 are transition relations.

Here, δ_1 is a total binary relation from $J \times invs \times V \times 2^J$ to $resps \times V$. It is used to map an invocation at the head of a particular *inv-buffer*, the current value for *val*, and the current set of failed processes, to a set of results, each of which consists of a new value for *val* and a response to be added to the corresponding *resp-buffer*. Further, δ_2 is a total binary relation from $V \times 2^J$ to V. It is used to map a value of *val* and the current set of failed processes to a set of new values.

Figure 7.1 presents the code for a canonical distributed service automaton, showing how these parameters are used.

We model a process failure at an endpoint i by an explicit input action $fail_i$. We say that a process i *fails* in a given execution if $fail_i$ occurs in that execution. We say that a process i is *correct* if $fail_i$ never occurs. For every process $i \in J$, we assume that the service has two tasks, which we call the i-perform and i-output tasks. The i-perform task includes the $perform_{i,k}$ action, which carries out operations invoked at endpoint i, and triggers a single response at endpoint i. The i-output task includes all the $b_{i,k}$ actions giving responses at i. The service may also include background processing tasks $compute_{g,k}$ ($g \in glob$), not related to any specific endpoint.

```
Canonical Service $S_k = (\mathcal{U}, J, k)$,
where $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$

Signature:

Inputs:
$a_{i,k}$, $a \in invs$, $i \in J$, the invocations at endpoint $i$
$fail_i$, $i \in J$

Outputs:
$b_{i,k}$, $b \in resps$, $i \in J$, the responses at endpoint $i$

Internals:
$perform_{i,k}$, $i \in J$
$compute_{g,k}$, $g \in glob$

State components:

$val \in V$, initially an element of $V_0$

$inv\text{-}buffer$, a mapping from $J$
    to finite sequences of $invs$,
    initially identically empty

$resp\text{-}buffer$, a mapping from $J$
    to finite sequences of $resps$,
    initially identically empty

$failed \subseteq J$, initially $\emptyset$

Transitions:

Input: $a_{i,k}$
Effect:
    add $a$ to end of $inv\text{-}buffer(i)$

Internal: $perform_{i,k}$, $i \in J$
Precondition:
    $a = head(inv\text{-}buffer(i))$
    $\delta_1((i, a, val, failed), (b, v))$
Effect:
    remove head of $inv\text{-}buffer(i)$
    $val \leftarrow v$
    add $b$ to end of $resp\text{-}buffer(i)$

Internal: $compute_{g,k}$, $g \in glob$
Precondition:
    $\delta_2((val, failed), v)$
Effect:
    $val \leftarrow v$

Output: $b_{i,k}$
Precondition:
    $b = head(resp\text{-}buffer(i))$
Effect:
    remove head of $resp\text{-}buffer(i)$

Input: $fail_i$
Effect:
    $failed \leftarrow failed \cup \{i\}$
    remove some invocations
        from $inv\text{-}buffer(i)$

Tasks:

For every $i \in J$:
    $i$-perform: $\{perform_{i,k}\}$
    $i$-output: $\{b_{i,k} : b \in resps\}$
For every $g \in glob$:
    $g$-compute: $\{compute_{g,k}\}$
```

Figure 7.1: A canonical distributed service $S_k = (\mathcal{U}, J, k)$

A canonical distributed service is thus allowed to perform rather flexible kinds of processing, both related and unrelated to individual endpoints.

The definition of fairness for I/O automata implies that, for every correct process $i \in J$, the service eventually responds to any outstanding invocation at i.

7.1.3 Failure detectors

Failure detectors, a class of distributed services, are of a special interest in this thesis. Informally, a distributed service is a failure detector if its responses depend only on information about failures. Formally, a service $S_k = (\mathcal{U}, J, k)$, where $\mathcal{U} = \langle V, V_0, invs, resps, glob, \delta_1, \delta_2 \rangle$, is a failure detector if the following properties are satisfied:

- $\mathit{invs} = \{\mathit{query}\}$, and
- $\forall i \in J$, action $\mathit{perform}_{i,k}$ does not modify the value of *val*, i.e., $\delta_1((i, a, v, \mathit{failed}), (v', b))$ only if $v = v'$.

Since query actions do not modify the value of *val*, it is convenient to define failure detectors in terms of traces from which all query actions are removed. We call these modified traces *failure detector histories.*

Strictly speaking, the failure detectors we define through automata do not provide all the functionality of those in the original model of [14]. Namely, because our failure detectors are automata, they cannot predict future actions. In the terminology of [19], our services encompass only *realistic* failure detectors.

In the following, we describe how a variety of well-known failure detectors [15, 14, 16, 32] can be modeled as distributed services. We give the definitions in our model of five canonical failure detectors from the literature, the perfect failure detector $\mathcal{P}$ [15], the eventually perfect failure detector $\diamond\mathcal{P}$ [15], the eventual leader failure detector Ω [14], the quorum failure detector Σ [20], and the failure signal failure detector $\mathcal{FS}$ [16, 32].

The perfect failure detector $\mathcal{P}$

The *perfect failure detector* $\mathcal{P}$ [15] outputs a subset of its endpoints — the set of *suspected* processes. It ensures the following properties:

Strong completeness: every failed process is eventually suspected at every correct process, and,

Strong accuracy: no process is suspected before it fails.

Formally, let J be the set of endpoints of $\mathcal{P}$. The internal state components in $\mathcal{P}$ are presented in Figure 7.2.

Components of *val*:
$\forall j \in J$: $S_j \subseteq J$, initially $\emptyset$

Figure 7.2: The composition of *val* in $\mathcal{P}$

The global task set $\mathit{glob} = \{g_j\}_{j \in J}$. For any $j \in J$, task g_j is responsible for maintaining the current set S_j of suspected processes for endpoint j.

Responses of $\mathcal{P}$ are of the form $\mathit{suspect}(J')$, $J' \subseteq J$. The set of internal state components in $\mathcal{P}$ is presented in Figure 7.3. Task i-output simply puts a $\mathit{suspect}(S_i)$ response into i's response buffer. For each $g_j \in \mathit{glob}$, task g_j-compute periodically updates S_i by adding recently failed processes.

The eventually perfect failure detector $\diamond\mathcal{P}$

The *eventually perfect failure detector* $\diamond\mathcal{P}$ [15] outputs a subset of its endpoints — the set of suspected processes. It ensures the following properties:

Strong completeness: every failed process is eventually suspected at every correct process, and

```
Internal: $perform_{i,k}$
Precondition:
    $query = head(inv\text{-}buffer(i))$
Effect:
    add $suspect(S_i)$ to $resp\text{-}buffer(i)$

Internal: $compute_{g_j,k}$, $j \in J$
Precondition:
    $failed - S_j \neq \emptyset$
Effect:
    $U \leftarrow$ choose a non-empty set in $failed - S_i$
    $S_j \leftarrow S_i \cup U$
```

Figure 7.3: Internal transitions in $\mathcal{P}$

Eventual strong accuracy: eventually, no process is suspected before it fails.

In other words, eventually $\diamond\mathcal{P}$ outputs the set of all failed processes at all correct endpoints.

Again, responses of $\diamond\mathcal{P}$ are of the form *suspect*(J'), $J' \subseteq J$. We model eventual perfection using a *mode* variable, which can take values *perfect* or *imperfect*. Initially, and after each new failure, *mode* is set to *imperfect*. A background task is responsible for eventually switching *mode* to *perfect*. Since failures must eventually stop, the *mode* eventually remains *perfect*. While in *perfect* mode, the failure detector suspects exactly the processes that have failed. In *imperfect* mode, any set of processes can be suspected. The set of internal state components in $\diamond\mathcal{P}$ is presented in Figure 7.4.

```
Components of val:
    $mode \in \{perfect, imperfect\}$, initially $imperfect$
    $oldfailed \subseteq J$, initially $\emptyset$
```

Figure 7.4: The composition of *val* in $\diamond\mathcal{P}$

The global task set $glob = \{g_1, g_2\}$. Task g_1 is responsible for, in case new failure occurs, setting *mode* to *imperfect* while task g_2 is responsible for setting *mode* back to *perfect*. Since eventually no new failure occurs, *mode* eventually stabilizes at *perfect*. The internal transitions of $\diamond\mathcal{P}$ are presented in Figure 7.5.

```
Internal: $perform_{i,k}$
Precondition:
    $query = head(inv\text{-}buffer(i))$
Effect:
    if $mode = perfect$ then
        add $suspect(failed)$ to $resp\text{-}buffer(i)$
    else
        choose $J' \subseteq J$
        add $suspect(J')$ to $resp\text{-}buffer(i)$

Internal: $compute_{g_1,k}$
Precondition:
    $failed \neq oldfailed$
Effect:
    $mode \leftarrow imperfect$
    $oldfailed \leftarrow failed$

Internal: $compute_{g_2,k}$
Precondition:
    $mode = imperfect$
Effect:
    $mode \leftarrow perfect$
```

Figure 7.5: Internal transitions in $\diamond\mathcal{P}$

The eventual leader failure detector Ω

The *eventual leader failure detector* Ω [14] provides *elect*(l) $(l \in J)$ responses at all endpoints. Eventually (assuming that not all processes fail), the latest *elect* announcements should be identical at all endpoints, and should indicate the name of a correct endpoint.

Components of *val*:
- $\mathit{oldfailed} \subseteq J$, initially $\emptyset$
- $\mathit{leader} \in J \cup \{\bot\}$, initially $\bot$

Figure 7.6: The composition of *val* in Ω

We use two global tasks g_1, g_2. Now g_1 sets *leader* to $\bot$ and removes any choice of leader, while g_2 chooses a leader by setting *leader* to some non-failed process. The corresponding transition definitions are presented in Figure 7.7.

```
Internal: $perform_{i,k}$
Precondition:
    $query = head(inv\text{-}buffer(i))$
Effect:
    if $leader \neq \bot$ then
        add $elect(leader)$ to $resp\text{-}buffer(i)$
    else
        choose $j \in J$
        add $elect(j)$ to $resp\text{-}buffer(i)$
```

```
Internal: $compute_{g_1,k}$
Precondition:
    $failed \neq oldfailed$
Effect:
    $leader \leftarrow \bot$
    $oldfailed \leftarrow failed$

Internal: $compute_{g_2,k}$
Precondition:
    $leader = \bot$
Effect:
    $leader \leftarrow$ choose $l \in J - failed$
```

Figure 7.7: Internal transitions in Ω

The quorum failure detector Σ

The *quorum failure detector* Σ [20] outputs a subset of its endpoints — the current *quorum.* Σ guarantees that, any two quorums (output at any processes and at any times) intersect, and, eventually, every quorum output at a correct process contains only correct processes.

It is convenient to define Σ with respect to a given *environment* $\mathcal{E}$, a set of proper subsets of J, the set of endpoints of Σ. $\mathcal{E}$ specifies subsets of processes that are allowed to fail in the same execution. For example $\mathcal{E} = \{S : S \subset J \wedge |S| < \lceil |J|/2 \rceil\}$ assumes that at most a minority of processes can fail, and $\mathcal{E} = \{S : S \subset J\}$ assumes that any proper subset of J can fail.

The set of internal state components in Σ is presented in Figure 7.8. Variables *mode* and *oldfailed* are as in the definition of $\diamond\mathcal{P}$. Variable Q contains the current quorum. Variable $\mathcal{Q}$ contains the list of all quorums produced so far.

Responses of Σ are of the form *quorum*(Q). Task i-output simply puts a *quorum*(Q) response into i's response buffer.

The global task set $glob = \{g_1, g_2, g_3\}$. As in the definition of $\diamond\mathcal{P}$, task g_1 is responsible for, in case a failure occurs, setting *mode* to *imperfect* while

```
Components of val:
    $mode \in \{perfect, imperfect\}$, initially $imperfect$
    $oldfailed \subseteq J$, initially $\emptyset$
    $Q \subseteq J$, initially $J$
    $\mathcal{Q}$, a set of subsets of $J$, initially $\emptyset$
```

Figure 7.8: The composition of *val* in Σ

task g_2 is responsible for setting *mode* back to *perfect*. Task g_3, described in Figure 7.9, computes the current quorum, based on the value of *mode*, the set of failed processes, environment $\mathcal{E}$, and all quorums computed earlier. More precisely, the current quorum is computed as a subset of J that intersects with any potential set of correct processes, and with any quorum computed earlier. Moreover, if $mode = perfect$, then the quorum must be a subset of $J - failed$. It is straightforward to see that any two computed quorums intersect, and, eventually, any quorum contains only correct processes.

```
Internal: $perform_{i,k}$
Precondition:
    $query = head(inv\text{-}buffer(i))$
Effect:
    add $quorum(Q)$ to $resp\text{-}buffer(i)$

Internal:
    $compute_{g_1,k}$ and $compute_{g_2,k}$ as for $\Diamond\mathcal{P}$

Internal: $compute_{g_3,k}$
Precondition:
    $true$
Effect:
    if $mode = perfect$ then
        $U \leftarrow J - failed$
    else
        $U \leftarrow J$
    $Q \leftarrow$ choose $V \subseteq U$ such that
        $(\forall S \in \mathcal{E},\ failed \subseteq S :\ V \cap (J - S) \neq \emptyset)$
        $\wedge\ (\forall S \in \mathcal{Q} :\ V \cap S \neq \emptyset)$
```

Figure 7.9: Internal transitions in Σ

Note that if environment $\mathcal{E}$ assumes that no more than a minority of processes in J can fail, then Q can be taken as any majority of processes (any majority in $J - failed$, if $mode = imperfect$).

On the other hand, if $\mathcal{E}$ assumes that any proper subset of processes in J can fail, then Q must include the set of all processes in $J - failed$ (exactly $J - failed$, if $mode = perfect$). In other words, in this environment, Σ must be at least as strong as the perfect failure detector $\mathcal{P}$ [19].

The failure signal failure detector $\mathcal{FS}$

The *failure signal* failure detector $\mathcal{FS}$ outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ outputs **green** at every process; after a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every non-faulty process [16, 32].

Formally, let J be the set of endpoints of $\mathcal{FS}$. The internal state components in $\mathcal{FS}$ are presented in Figure 7.10.

The global task set $glob = \{g_j\}_{j \in J}$. For any $j \in J$, task g_j is responsible for

Components of *val*:
$\forall j \in J$: $s_j \in \{\mathbf{green}, \mathbf{red}\}$, initially **green**

Figure 7.10: The composition of *val* in $\mathcal{FS}$

maintaining the signal (**green** or **red**) for endpoint j.

Responses of $\mathcal{FS}$ are of the form $\mathit{signal}(s)$, $s \in \{\mathbf{green}, \mathbf{red}\}$. The set of internal state components in $\mathcal{FS}$ is presented in Figure 7.11. $\delta_1(i, \mathit{query}, \mathit{failed})$ puts a $\mathit{signal}(s_i)$ response into i's response buffer. δ_2 sets s_i to **red** in case a failure occurs.

```
Internal: $perform_{i,k}$
Precondition:
    $query = head(inv\text{-}buffer(i))$
Effect:
    add $signal(s_i)$ to $resp\text{-}buffer(i)$

Internal: $compute_{g_j,k}$, $j \in J$
Precondition:
    $failed \neq \emptyset \wedge s_j = \mathbf{green}$
Effect:
    $s_j \leftarrow \mathbf{red}$
```

Figure 7.11: Internal transitions in $\mathcal{FS}$

7.1.4 Reliable channels

As in [57], we model message-passing systems by introducing, for every two processes i and j, a reliable channel automaton $C_{i,j}$. Let M be a fixed message alphabet. The inputs of $C_{i,j}$ are $\mathit{send}(m)_{i,j}$ ($m \in M$) actions, and its outputs are $\mathit{receive}(m)_{i,j}$ ($m \in M$) actions. $C_{i,j}$ has exactly two endpoints i and j and a task partition, ensuring that every message sent from i to j is eventually received.

7.1.5 Atomic objects

In this section, we define another special class of distributed services, called *atomic objects.* A service S_k of type $\mathcal{U}_k = \langle V, V_0, \mathit{invs}, \mathit{resps}, \mathit{glob}, \delta_1, \delta_2 \rangle$ is an atomic object if $\mathcal{U}_k$ is *sequential* which means:

- $\mathit{glob} = \emptyset$, i.e., there are no background processing tasks.
- $\forall f_1, f_2 \in 2^J$, $\delta_1((i, a, v, f_1), (b, v'))$ only if $\delta_1((i, a, v, f_2), (b, v'))$, i.e., the response on any invocation does not depend on failures.

That is, unlike failure detectors, the output of atomic objects is not allowed to include information about failures.

We say that $\mathcal{U}_k$ is *oblivious* if no response depends on the endpoint that the corresponding invocation is applied on, i.e., $\forall i, j \in J$, $\delta_1((i, a, v, f), (b, v))$ only if $\delta_1((j, a, v, f), (b, v))$.

We say that $\mathcal{U}_k$ is *deterministic* if V_0 is a singleton set $\{v_0\}$, and δ_1 is a mapping, that is, for every $(i, a, v, f) \in J \times \mathit{invs} \times V \times 2^J$, there is *exactly one* $(b, v') \in \mathit{resps} \times V$ such that $\delta_1((i, a, v, f), (b, v'))$.

Examples:

register: Here, V is any set, $V_0 = \{v_0\}$ where $v_0 \in V$, $invs = \{read\} \cup \{write(v) : v \in V\}$, $resps = V \cup \{ack\}$, and

$$\begin{aligned}\delta_1 = \{((i, read, v, f), (v, v)) : i, \in J, v \in V, f \in 2^J\} \cup \\ \{((i, write(v), v', f), (ack, v)) : i \in J, v, v' \in V, f \in 2^J\}.\end{aligned}$$

consensus: Here, $V = \{\{0\}, \{1\}, \perp\}$, $V_0 = \{\perp\}$, $invs = \{init(v) : v \in \{0, 1\}\}$, $resps = \{decide(v) : v \in \{0, 1\}\}$, and

$$\begin{aligned}\delta_1 = \{((i, init(v), \perp, f), (decide(v), \{v\})) : i \in J, v \in V, f \in 2^J\} \cup \\ \{((i, init(v), \{v'\}, f), (decide(v'), \{v'\})) : i \in J, v, v' \in V, f \in 2^J\}.\end{aligned}$$

7.1.6 Distributed system model

Let $\Pi = \{p_1, p_2, \ldots, p_n\}$ be the set of processes. We model a distributed system as a parallel composition of *process automata* and a collection of distributed services (Figure 7.12). The process automata handle requests from the *external world*, and communicate through invoking operations on and receiving responses from the distributed services. The distributed services do not directly communicate with one another, but may interact indirectly via processes. The distributed services might include point-to-point reliable channels, wait-free atomic objects and failure detectors.

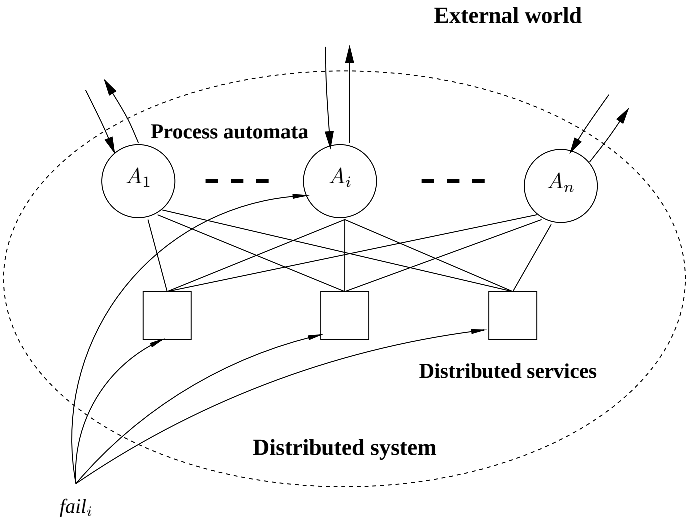

Figure 7.12: System model

Problems

A *problem* $\mathcal{M}$ is a predicate on traces of a distributed system. In particular, $\mathcal{M}$ specifies the interface between the processes and the external world in terms of $\mathcal{M}.invs$ and $\mathcal{M}.resps$, operations which may be invoked on the processes by their external *clients* and responses which may be returned by the processes to their external clients, respectively.

Process automata

A_i, $i = 1, 2, \dots, p_n$, denotes the process automata on which process p_i runs an algorithm A. Each process automaton A_i has the following inputs and outputs:

- Inputs a_i, $a \in \mathcal{M}.invs$, and outputs b_i, $b \in \mathcal{M}.resps$, where $\mathcal{M}$ is the specification of the problem our system is intended to solve.

- For each service $S_k = (\mathcal{U}_k, J_k, k)$ such that $i \in J_k$, outputs $a_{i,k}$, $a \in \mathcal{U}_k.invs$, and inputs $b_{i,k}$, $b \in \mathcal{U}_k.resps$.

- Input $fail_i$.

We assume that, for each $p_i \in \Pi$, A_i is deterministic (we do not consider randomized algorithms), and the locally-controlled actions of A_i are partitioned in a finite set of tasks. We assume that the $fail_i$ input action affects A_i in such a way that, from that point onward, no output actions are enabled, i.e., we consider only crash failures.

Complete system

Let A be an algorithm, $\mathcal{K}$ be any set of natural numbers and $\mathcal{D}$ be a failure detector ($\mathcal{D} \notin \{S_k\}_{k\in\mathcal{K}}$). The complete system $\langle A, \mathcal{K}, \mathcal{D}\rangle$ of processes in Π running algorithm A, and using services $\{S_k\}_{k\in\mathcal{K}}$ is a parallel composition of $\{A_i\}_{p_i\in\Pi}$, $\{S_k\}_{k\in\mathcal{K}}$ and $\mathcal{D}$, where all the actions used to communicating among the components are hidden.

Let τ be any trace of a distributed system $\langle A, \mathcal{K}, \mathcal{D}\rangle$. We denote by $failed(\tau)$ the set of processes that fail in τ: $failed(\tau) = \{i \in \Pi : fail_i \in \tau\}$.

Let $\mathcal{E}$ be any environment, i.e., any set of proper subsets of Π. Recall that $\mathcal{E}$ specifies subsets processes that are allowed to fail in the same execution. (This definition of environments is less fine-grained than the one of [14], because it does not employ the timing of failures.) We say that τ is *$\mathcal{E}$-compliant* if $failed(\tau) \in \mathcal{E}$.

Solving a problem

We say that a distributed system $\langle A, \mathcal{K}, \mathcal{D}\rangle$ solves a problem $\mathcal{M}$ in an environment $\mathcal{E}$ if the set of fair $\mathcal{E}$-compliant traces of $\langle A, \mathcal{K}, \mathcal{D}\rangle$ satisfies $\mathcal{M}$.

In the following, we fix the set $\mathcal{K}$ of available distributed services, and we say that a failure detector $\mathcal{D}$ solves a problem $\mathcal{M}$ in an environment $\mathcal{E}$ if there exists a distributed system $\langle A, \mathcal{K}, \mathcal{D}\rangle$ that solves $\mathcal{M}$ in $\mathcal{E}$.

Implementing distributed services

Implementing a distributed service is just a special case of solving a problem. We say that a distributed system $\langle A, \mathcal{K}, \mathcal{D}\rangle$ *implements* a service S_k in an environment $\mathcal{E}$ if any fair $\mathcal{E}$-compliant trace of $\langle A, \mathcal{K}, \mathcal{D}\rangle$ is a trace of S_k.

Respectively, we say that a failure detector $\mathcal{D}$ implements a service S_k in $\mathcal{E}$ if there exists a distributed system $\langle A, \mathcal{K}, \mathcal{D}\rangle$ that implements S_k in $\mathcal{E}$.

7.1.7 A weakest failure detector

The definition of the weakest failure detector to solve a given problem in a given environment is now straightforward.

We say that $\mathcal{D}$ is *weaker than* $\mathcal{D}'$ in $\mathcal{E}$ if $\mathcal{D}'$ implements $\mathcal{D}$ in $\mathcal{E}$. We say that $\mathcal{D}$ is *the weakest failure detector to solve a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ if:

(1) $\mathcal{D}$ is *sufficient to solve* $\mathcal{M}$ *in* $\mathcal{E}$, i.e., there exists a distributed system $\{\Pi, \mathcal{K}, \mathcal{D}\}$ that solves $\mathcal{M}$ in $\mathcal{E}$.

(2) $\mathcal{D}$ is *necessary to solve* $\mathcal{M}$ *in* $\mathcal{E}$, i.e., if a failure detector $\mathcal{D}'$ is sufficient to solve $\mathcal{M}$ in $\mathcal{E}$, then $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$.

There may be a number of failure detectors that satisfy these conditions. So it would be more precise to say "*a* weakest failure detector".

7.1.8 Applications

We conjecture that the tight lower bounds on the amount of synchrony presented in this thesis hold also in the new model discussed in this chapter. In particular, the proofs of Chapters 3–6 can be transformed in a straightforward manner to be valid in the I/O automata-based model.

In particular, the construction of simulation trees, based on sampling the output of a failure detector $\mathcal{D}$ (Chapters 3, 5 and 6), does not depend on the way $\mathcal{D}$ is modeled. Recall that a crucial assumption about the output of $\mathcal{D}$ used in the construction is transitivity of the simulation stimuli: if a DAG G_p includes a path $[q, d, k] \rightarrow [q', d', k'] \rightarrow [q'', d'', k'']$ (a simulation stimulus), then $[q, d, k] \rightarrow [q'', d'', k'']$ is also a path in G_p. Thus, by only observing the output of its failure detector module, q'' cannot conclude whether q' has previously taken a step in which q' has seen value d', or not. More generally, the output of a failure detector module provides some hints about the current failure detector history, but it cannot leak any information on the *steps* of other processes. This property is trivially satisfied by failure detector services described in this chapter: a query action cannot modify the internal state of a failure detector, and, thus, no future query action can keep track of it.

The careful adaptation of the results of this thesis to the alternative model is the subject of further research.

7.2 Future directions

The failure detector research domain initiated a number of interesting results. One thread of research has concentrated on determining the weakest failure detectors for solving various problems in various environment (for example, [14, 5, 30, 2, 21]). A second has focused on practical aspects of implementing failure detectors and designing failure detector-based algorithms (for example, [17, 4, 27, 3]). However, there still seems to be a lot of space for further research. In Sections 4.9 and 6.6, we discuss a few open questions related to the results of this thesis.

Determining the minimal synchrony assumptions for a given problem is just a special case of a more general question: what is the strongest *adversary* that allows us to solve the problem. Adversaries can be parameterized according to their abilities to (a) schedule communication and processing, (b) remove or corrupt communication messages, (c) enforce Byzantine behavior of processes, etc.

It would be very appealing to develop techniques for (1) determining the strongest adversaries for important synchronization problems, and (2) designing algorithms that enjoy high performance facing an adversary of a given power.

7.3 Summary

The impossibility results of [28, 24, 56] establish that asynchronous distributed computability is essentially different from classical Turing computability [69]: the consensus problem, trivially solvable on a single processor, is impossible to solve in an asynchronous system of two or more processors of which one can fail by crashing. One way to circumvent this impossibility is to augment the asynchronous system model with certain synchrony assumptions. The notion of the weakest failure detector [14] proposes a way to describe minimal synchrony assumptions that are sufficient for solving distributed computing problems.

This thesis has focused on determining the weakest failure detectors for three fundamental problems in distributed computing: solving fault-tolerant mutual exclusion, solving non-blocking atomic commit, and boosting the synchronization power of atomic objects.

Bibliography

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1 – 20, February 1991.

[2] M. K. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM J. Comput.*, 29(6):2040–2073, 2000.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 328–337, 2004.

[4] M. K. Aguilera, G. Le Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, pages 354–370, 2002.

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

[6] P. Attie, N. A. Lynch, and S. Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical report, MIT Laboratory for Computer Science, MIT-LCS-TR-877, 2002.

[7] P. C. Attie, R. Guerraoui, P. Kouznetsov, N. A. Lynch, and S. Rajsbaum. The impossibility of boosting distributed service resilience. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, June 2005. Available at http://theory.lcs.mit.edu/tds/papers/Attie/boosting-tr.ps.

[8] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[9] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.

[10] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense! In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 363–372, August 1994.

[11] T. D. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. The h_m^r hierarchy is not robust. Manuscript.

[12] T. D. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 334–343, August 1994.

[13] T. D. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM J. Comput.*, 34(2):333–357, 2004.

[14] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[15] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[16] B. Charron-Bost and S. Toueg. Unpublished notes, 2001.

[17] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Computers*, 51(1& 5):13–32 & 561–580, 2002.

[18] B. A. Coan and J. L. Welch. Transaction commit in a realistic timing model. *Distributed Computing*, 4(2):87–103, 1990.

[19] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *Proceedings of the IEEE Symposium on Dependable Systems and Networks (DSN 2002)*, Washington DC, June 2002.

[20] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs message passing. Technical Report IC/2003/77, EPFL, December 2003. Available at http://icwww.epfl.ch/publications/.

[21] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 338–346, July 2004.

[22] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Dustributed Computing (JPDC)*, 65(4):492–505, April 2005.

[23] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[24] D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

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

[26] J. Eisler, V. Hadzilacos, and S. Toueg. The quorum failure detector and its relation to consensus and registers. Unpublished note, June 25 2004.

[27] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Computers*, 52(2):99–112, 2003.

[28] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.

[29] M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein, and A. Smith. Detectable byzantine agreement secure against faulty majorities. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[30] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 470–477, 1999.

[31] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *LNCS*, pages 393–481. Springer-Verlag, 1978.

[32] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15:17–25, January 2002.

[33] R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, pages 461–473, August 2002.

[34] R. Guerraoui and P. Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, October 2003.

[35] R. Guerraoui and P. Kouznetsov. The weakest failure detector for non-blocking atomic commit. Technical Report IC/2003/47, EPFL, May 2003. Available at http://icwww.epfl.ch/publications/.

[36] R. Guerraoui and P. Kouznetsov. The gap in circumventing the consensus impossibility. Technical report, EPFL, ID:IC/2004/28, January 2004. Available at http://icwww.epfl.ch/publications/.

[37] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. *Fault-Tolerant Distributed Computing*, pages 201–208, 1987.

[38] V. Hadzilacos. A note on group mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 100–106, August 2001.

[39] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcast and related problems. Technical report, Cornell University, Computer Science, May 1994.

[40] V. Hadzilacos and S. Toueg. The weakest failure detector to solve quittable consensus and non-blocking atomic commit. Unpublished manuscript – private communication, May 2003.

[41] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[42] M. Herlihy and E. Ruppert. On the existence of booster types. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 653–663, 2000.

[43] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[44] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993.

[45] P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.

[46] P. Jayanti and S. Toueg. Some results on the impossibility, universality and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, volume 647 of *LNCS*. Springer Verlag, 1992.

[47] Y.-J. Joung. Asynchronous group mutual exclusion. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, June 1998.

[48] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685, July 2001.

[49] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

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

[51] L. Lamport. The mutual exclusion problem. Parts I&II. *Journal of the ACM*, 33(2):313–348, April 1986.

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

[53] W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, volume 857 of *LNCS*, pages 280–295. Springer Verlag, 1994.

[54] W.-K. Lo and V. Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal of Computing*, 30(3):689–728, 2000.

[55] S. Lodha and A. D. Kshemkalyani. A fair distributed mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):537–549, June 2000.

[56] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.

[57] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[58] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[59] D. Manivannan and M. Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 525–530, October 1994.

[60] A. Mostefaoui, M. Raynal, and F. Tronel. From Binary Consensus to Multivalued Consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5–6):207–212, March 2000.

[61] G. Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 100–109, August 1995.

[62] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, Massachusetts, 1986.

[63] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[64] E. Ruppert. Determining consensus numbers. *SIAM Journal of Computing*, 30(4):1156–1168, 2000.

[65] E. Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 279, 1997.

[66] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[67] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, May 1993.

[68] D. Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California at Berkeley, May 1982. Technical Memorandum UCB/ERL M82/45.

[69] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, Second Series*, pages 230–265, November 1936.

[70] K. Vidyasankar. A simple group mutual l-exclusion algorithm. *Information Processing Letters*, 85:79–85, 2003.

[71] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Symposium on Foundations of Computer Science*, pages 233–246, 1986.

List of Figures

3.1 Building a DAG: process p 20
3.2 A DAG and a tree 22
3.3 A fork and a hook 24
3.4 Locating a decision gadget 25
3.5 Locating a fork (Case 1) or a hook (Case 2) 26
3.6 Extracting a correct leader: code for each process p 27
3.7 Extracting Σ: code for each process p 30

4.1 Failure detection scenario for $\mathcal{T}$ 35
4.2 Reduction algorithm $T_{\mathcal{D}\to\mathcal{T}}$: code for each process i 38
4.3 FTME algorithm using $\mathcal{T}$: code for each process i 41
4.4 FTGME algorithm using $\mathcal{T}$: code for each process i 49
4.5 Comparative performance analysis of mutual exclusion algorithms 51

5.1 Transforming binary QC into multivalued QC: code for each process p 56
5.2 Using Ψ to solve QC 57
5.3 Extracting Ψ from $\mathcal{D}$ and QC algorithm $\mathcal{A}$: code for each process p 58
5.4 Illustration of proof of Lemma 5.6 63
5.5 Using $\mathcal{FS}$ to transform QC into NBAC: code for each process p . . 67
5.6 Transforming NBAC into QC: code for each process p 67

6.1 A team consensus algorithm: code for each process p 76
6.2 Building a DAG: code for each process p 78
6.3 A fork, a hook, and a rake 81
6.4 Locating a rake in $\Upsilon^{P,l}$ 82
6.5 Extracting Ω_n: code for each process p 88

7.1 A canonical distributed service $S_k = (\mathcal{U}, J, k)$ 98
7.2 The composition of *val* in $\mathcal{P}$ 99
7.3 Internal transitions in $\mathcal{P}$ 100
7.4 The composition of *val* in $\diamond\mathcal{P}$ 100

7.5 Internal transitions in $\Diamond\mathcal{P}$ 100

7.6 The composition of *val* in Ω 101

7.7 Internal transitions in Ω 101

7.8 The composition of *val* in Σ 102

7.9 Internal transitions in Σ 102

7.10 The composition of *val* in $\mathcal{FS}$ 103

7.11 Internal transitions in $\mathcal{FS}$ 103

7.12 System model 104

Curriculum Vitae

Petr Kouznetsov was born on July 23, 1974, in Leningrad, USSR, converted later into Saint-Petersburg, Russia. He received B.Sc. and M.Sc. in mathematics, *cum laude*, from Saint-Petersburg State Institute of Fine Mechanics and Optics (Technical University) in 1995 and 1997, respectively. In 1994–1996, he participated in a Russian-Dutch research project “Nonlinear problems in resonance”; in early 1996, he visited Delft University of Technology as a guest researcher. In 1997–1999, he worked as a systems engineer in the area of computer networking for BCC, Saint-Petersburg. In 1999, he enrolled for the Communication Systems Doctoral School of Ecole Polytechnique Fédérale de Lausanne (EPFL). Since 2000, he has been working as a research assistant and, since 2001, a Ph.D. student in the Operating Systems Laboratory of EPFL, and thereafter in the newly formed Distributed Programming Laboratory of EPFL, under the supervision of Prof. Rachid Guerraoui.