Unreliable Failure Detectors for Asynchronous Distributed Systems

Tushar Deepak Chandra
Ph.D Thesis

93-1377
August 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

UNRELIABLE FAILURE DETECTORS FOR ASYNCHRONOUS DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tushar Deepak Chandra

May 1993

UNRELIABLE FAILURE DETECTORS FOR ASYNCHRONOUS DISTRIBUTED SYSTEMS

Tushar Deepak Chandra, Ph.D.

Cornell University 1993

It is well-known that several fundamental problems of fault-tolerant distributed computing, such as *Consensus* and *Atomic Broadcast*, cannot be solved in asynchronous systems with crash failures. These impossibility results stem from the lack of reliable failure detection in such systems. To circumvent such impossibility results, we introduce the concept of *unreliable failure detectors* that can make mistakes, and study the problem of using them to solve Consensus (and Atomic Broadcast).

It is easy to solve Consensus using a "perfect" failure detector (one that does not make mistakes). But is perfect failure detection necessary to solve Consensus? We show that Consensus is solvable with unreliable failure detectors, even if they make an infinite number of mistakes. This leads to the following question: What is the "weakest" failure detector for solving Consensus? We introduce a notion of algorithmic reducibility that allows us to compare seemingly incomparable failure detectors. Using this concept, we show that one of the failure detectors that

we introduce here is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes.

We also show that Consensus and Atomic Broadcast are equivalent in asynchronous systems. Thus all our results regarding the solvability of Consensus using failure detectors, apply to Atomic Broadcast as well.

The work in this thesis was funded by an IBM graduate fellowship and grants from NSF, DARPA/NASA, the IBM Endicott Programming Laboratory, Siemens Corp and the Natural Siences and Engineering Research Council of Canada.

Biographical Sketch

Tushar Deepak Chandra was born in New Delhi, India on November 13, 1966. He spent his childhood in various cities in India: Bombay, Calcutta and finally Kanpur. After completing high school at the Doon school, he went on to do a Bachelor of Technology in Computer Science at the Indian Institute of Technology at Kanpur. He joined the graduate program in Computer Science at Cornell University in August 1988.

This thesis is dedicated to my parents who taught me how to think.

Acknowledgements

A large number of people contributed either directly or indirectly to this thesis. I was extremely fortunate to have Sam Toueg as my advisor. Without his guidance, I would still be trying to shave off deltas from broadcast algorithms; without his cooking knowledge, I would never have learnt to cook pasta sauce "a la putanesca". Sam has put in as much work—if not more—towards this thesis, as I have.

I am indebted to Vassos Hadzilacos, my other co-author, for his invaluable contributions to this work. Without his help, Chapter 4 would not have happened. Further, his critical readings of the work presented in Chapter 3 have greatly influenced the way it is presented.

The idea of using unreliable failure detectors came from the many invaluable discussions I had with the Isis folks, in particular Aleta Ricciardi and Ken Birman (also see [RB91]). Somnath Biswas, Dexter Kozen, George Odifreddi and Rod Downey taught me recursion theory and thus gave me the tools to prove the result in Chapter 4. I was introduced to the rotating coordinator paradigm by a homework solution of Navin Budhiraja (see Figure 3.6). Prasad Jayanti greatly simplified the algorithm in Figure 3.3. Earlier, there were two algorithms in place of this algorithm and twice the text!

I would like to thank my thesis committee, i.e. Ken Birman, Dexter Kozen, Anil Nerode, and Sam Toueg for their guidance and support. I would also like to thank Özalp Babaoğlu, Cynthia Dwork, Keith Marzullo, Gil Neiger, Mike Reiter, Fred Schneider and the distributed systems group at Cornell for their comments and criticisms.

A special thanks to my various housemates, V. V. Bhaskar, Sandip Bose, Navin Budhiraja, Suresh Chari, Radha Jagadeesan, "Junglee" Jayesh, Mayan Moudgill, and Alka Parikh for the many stimulating discussions that have influenced me and hence this thesis.

Finally, I'd like to thank my family: Romesh, Manjula, and Siddharth Chandra for putting up with erratic hours, forgetfulness, etc. and for providing me with their love and support.

Table of Contents

List of Figures

Chapter 1

Introduction

The design and verification of fault-tolerant distributed applications is widely viewed as a complex endeavour. In recent years, several paradigms have been identified which simplify this task. Key among these are *Consensus* and *Atomic Broadcast*. Roughly speaking, Consensus allows processes to reach a common decision, which depends on their initial inputs, despite failures. Consensus algorithms can be used to solve many problems that arise in practice, such as electing a leader or agreeing on the value of a replicated sensor. Atomic Broadcast allows processes to reliably broadcast messages, so that they agree on the set of messages they deliver and the order of message deliveries. Applications based on these paradigms include SIFT [WLG+78], State Machines [Lam78,Sch90], Isis [BJ87, BCJ+90], Psync [PBS89], Amoeba [Mul87], Delta-4 [Pow91], Transis [ADKM91], HAS [Cri87], FAA [CDD90], and Atomic Commitment.

Given their wide applicability, Consensus and Atomic Broadcast have been extensively studied by both theoretical and experimental researchers for over a decade. In this thesis, we focus on solutions to Consensus and Atomic Broadcast in the asynchronous model of distributed computing. Informally, a distributed system is *asynchronous* if there is no bound on message delay, clock drift, or the time necessary to execute a step. Thus, to say that a system is asynchronous is to make *no* timing assumptions whatsoever. This model is attractive and has recently gained much currency for several reasons: It has simple semantics; applications programmed on the basis of this model are easier to port than those incorporating specific timing assumptions; and in practice, variable or unexpected workloads are sources of asynchrony—thus synchrony assumptions are at best probabilistic.

Although the asynchronous model of computation is attractive for the reasons outlined above, it is well known that Consensus and Atomic Broadcast cannot be solved deterministically in an asynchronous system that is subject to even a single *crash* failure [FLP85,DDS87].[1] Essentially, the impossibility results for Consensus and Atomic Broadcast stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow".

To circumvent these impossibility results, previous research focused on the use of randomization techniques [CD89], the definition of some weaker problems and their solutions [DLP+86,ABD+87,BW87,BMZ88], or the study of several models of *partial synchrony* [DDS87,DLS88]. Nevertheless, the impossibility of deterministic solutions to many agreement problems (such as Consensus and Atomic Broadcast) remains a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed computing.

In this thesis, we propose an alternative approach to circumvent such impossibility results, and to broaden the applicability of the asynchronous model of computation. Since impossibility results for asynchronous systems stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow", we propose to augment the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, we model the concept of *unreliable failure detectors* for systems with *crash* failures.

We consider *distributed* failure detectors: each process has access to a local

[1] Roughly speaking, a crash failure occurs when a process that has been executing correctly, stops prematurely. Once a process crashes, it does not recover.

failure detector module. Each local module monitors a subset of the processes in the system, and maintains a list of those that it currently suspects to have crashed. We assume that each failure detector module can make mistakes by erroneously adding processes to its list of suspects: i.e, it can suspect that a process p has crashed even though p is still running. If this module later believes that suspecting p was a mistake, it can remove p from its list. Thus, each module may repeatedly add and remove processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects.

It is important to note that the mistakes made by an unreliable failure detector should not prevent any correct process from behaving according to specification even if that process is (erroneously) suspected to have crashed by all the other processes. For example, consider an algorithm that uses a failure detector to solve Atomic Broadcast in an asynchronous system. Suppose all the failure detector modules wrongly (and permanently) suspect that correct process p has crashed. The Atomic Broadcast algorithm must still ensure that p delivers the same set of messages, in the same order, as all the other correct processes. Furthermore, if p broadcasts a message m, all correct processes must deliver m.[2]

We define failure detectors in terms of *abstract* properties as opposed to giving specific *implementations*; the hardware or software implementation of failure detectors is not the concern of this thesis. This approach allows us to design applications and prove their correctness relying solely on these properties, without referring to low-level network parameters (such as the exact duration of time-outs that are used to implement failure detectors). This makes the presentation of applications and their proof of correctness more modular. Our approach is well-suited to model many existing systems that decouple the design of fault-tolerant applications from

[2]A different approach was taken by the Isis system [RB91]: a correct process that is wrongly suspected to have crashed, is forced to leave the system. In other words, the Isis failure detector forces the system to conform to its view. To applications such a failure detector makes no mistakes. For a more detailed discussion on this, see Section 5.3.

the underlying failure detection mechanisms, such as the *Isis Toolkit* [BCJ+90] for asynchronous fault-tolerant distributed computing.

We characterize a failure detector by specifying the *completeness property* and *accuracy property* that it must satisfy. Informally, *completeness* requires that the failure detector eventually suspects every process that actually crashes,[3] while *accuracy* restricts the mistakes that a failure detector can make. We define two completeness and four accuracy properties, which gives rise to eight failure detectors, and consider the problem of solving Consensus using each one of them.[4]

To do so, we first introduce the concept of "reducibility" among failure detectors. Informally, a failure detector $\mathcal{D}'$ *is reducible to failure detector* $\mathcal{D}$ if there is a distributed algorithm $T_{\mathcal{D} \to \mathcal{D}'}$ that can use $\mathcal{D}$ to emulate $\mathcal{D}'$. Given this reduction algorithm, anything that can be done using failure detector $\mathcal{D}'$, can be done using $\mathcal{D}$ instead. Two failure detectors are *equivalent* if they are reducible to each other. Using this concept, we partition our eight failure detectors into four equivalence classes, and consider how to solve Consensus for each class.

We show that only four of our eight failure detectors can be used to solve Consensus in systems in which any number of processes may crash. However, if we assume that a majority of the processes do not crash, then any of our eight failure detectors can be used to solve Consensus. In order to better understand where the majority requirement becomes necessary, we study an infinite hierarchy of failure detectors that contains the eight failure detectors mentioned above, and show exactly where in this hierarchy the majority requirement becomes necessary.

Of special interest is $\diamond\mathcal{W}$, the weakest failure detector considered in this thesis. Informally, $\diamond\mathcal{W}$ satisfies the following two properties:

[3]In this introduction, we say that the failure detector suspects that a process p has crashed if *any* local failure detector module suspects that p has crashed.

[4]We later show that Consensus and Atomic Broadcast are *equivalent* in asynchronous systems: any Consensus algorithm can be transformed into an Atomic Broadcast algorithm and vice versa. Thus, we can focus on Consensus since all our results will automatically apply to Atomic Broadcast as well.

- *Completeness*: There is a time after which every process that crashes is always suspected by some correct process.

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

The failure detector $\diamond\mathcal{W}$ can make an infinite number of mistakes: Each local failure detector module of $\diamond\mathcal{W}$ can repeatedly add and then remove *correct* processes from its list of suspects (this reflects the inherent difficulty of determining whether a process or a link is just slow or whether it has crashed). Moreover, some correct processes may be erroneously suspected to have crashed by all the other processes throughout the entire execution.

The two properties of $\diamond\mathcal{W}$ state that eventually something must hold forever; this may appear too strong a requirement to implement in practice. However, when solving a problem that "terminates", such as Consensus, it is not really required that the properties hold *forever*, but merely that they hold for a *sufficiently long time*, i.e., long enough for the algorithm that uses the failure detector to achieve its goal. For instance, in practice our algorithm that solves Consensus using $\diamond\mathcal{W}$ only needs the two properties of $\diamond\mathcal{W}$ to hold for a relatively short period of time. However, in an asynchronous system it is not possible to quantify "sufficiently long", since even a single process step or a single message transmission is allowed to take an arbitrarily long amount of time. Thus, it is convenient to state the properties of $\diamond\mathcal{W}$ in the stronger form given above.

Another advantage of using $\diamond\mathcal{W}$ (as opposed to stronger failure detectors) is the following. Consider an application that relies on $\diamond\mathcal{W}$ for its correctness. If this application is run in a system in which the failure detector "malfunctions" and fails to meet the specification of $\diamond\mathcal{W}$, then we may lose the *liveness* properties of the application, but its *safety* properties will never be violated.

The failure detector abstraction is a clean extension to the asynchronous model of computation that allows us to solve many problems that are otherwise unsolvable. Naturally, the question arises of how to support such an abstraction in an actual system. Since we specify failure detectors in terms of abstract properties, we are not committed to a particular implementation. For instance, one could envision specialised hardware to support this abstraction. However, most implementations of failure detectors are based on time-out mechanisms. For the purpose of illustration, we now outline one such implementation of $\diamond\mathcal{W}$.

Informally, if a process times-out on some process q, it adds q to its list of suspects, and it broadcasts a message to all processes (including q) with this information. Any process that receives this broadcast adds q to its list of suspects. If q has not crashed, it broadcasts a refutation. If a process receives q's refutation, it removes q from its list of suspects.

In the *purely* asynchronous system, this scheme does not implement $\diamond\mathcal{W}$:[5] an unbounded sequence of premature time-outs (with corresponding refutations) may cause every correct process to be repeatedly added and then removed from every correct process' list of suspects, thereby violating the accuracy property of $\diamond\mathcal{W}$. Nevertheless, in many practical systems, one can choose the time-out periods so that eventually there are no premature time-outs on at least one correct process p. This gives the accuracy property of $\diamond\mathcal{W}$: there is a time after which p is permanently removed from all the lists of suspects. Recall that, in practice, it is not necessary for this to hold permanently; it is sufficient that it holds "long enough" for the application using the failure detector to complete its task. Accordingly, it is not necessary for the premature time-outs on p to cease permanently: it is sufficient that they cease for "long enough".

Having made the point that $\diamond\mathcal{W}$ can be implemented in practical systems using time-outs, we reiterate that all reasoning about failure detectors (and algorithms that use them) should be done in terms of their abstract properties and not in terms

[5]Indeed, no scheme could implement $\diamond\mathcal{W}$ in the purely asynchronous system: as we show in Section 3.5.2, such an implementation could be used to solve Consensus in such a system, contradicting the impossibility result of [FLP85].

of any particular implementation. This is an important feature of this approach, and the reader should refrain from thinking of failure detectors in terms of specific time-out mechanisms.

The failure detection properties of $\diamond\mathcal{W}$ are *sufficient* to solve Consensus in asynchronous systems. But are they *necessary*? For example, consider failure detector $\mathcal{A}$ that satisfies the completeness property of $\diamond\mathcal{W}$ and the following weakening of $\diamond\mathcal{W}$'s accuracy property:

- *Accuracy*: There is a time after which some correct process is never suspected by at least 99% of the correct processes.

$\mathcal{A}$ is clearly weaker than $\diamond\mathcal{W}$. Is it possible to solve Consensus using $\mathcal{A}$? Indeed what is the *weakest* failure detector *sufficient* to solve Consensus in asynchronous systems? In trying to answer this fundamental question we run into a problem. Consider failure detector $\mathcal{B}$ that satisfies the following two properties:

- *Completeness*: There is a time after which every process that crashes is always suspected by *all* correct processes.

- *Accuracy*: There is a time after which some correct process is never suspected by a majority of the processes.

It seems that $\mathcal{B}$ and $\diamond\mathcal{W}$ are *incomparable*: $\mathcal{B}$'s completeness property is stronger than $\diamond\mathcal{W}$'s, and $\mathcal{B}$'s accuracy property is weaker than $\diamond\mathcal{W}$'s. Is it possible to solve Consensus in an asynchronous system using $\mathcal{B}$? The answer turns out to be "yes" (provided that this asynchronous system has a majority of correct processes, as $\diamond\mathcal{W}$ also requires). Since $\diamond\mathcal{W}$ and $\mathcal{B}$ appear to be incomparable, one may be tempted to conclude that $\diamond\mathcal{W}$ cannot be the "weakest" failure detector with which Consensus is solvable. Even worse, it raises the possibility that no such "weakest" failure detector exists.

However, a closer examination reveals that $\mathcal{B}$ and $\diamond\mathcal{W}$ are indeed comparable: there is a distributed algorithm $T_{\mathcal{B}\rightarrow\diamond\mathcal{W}}$ that can transform $\mathcal{B}$ into a failure detector with the properties of $\diamond\mathcal{W}$. $T_{\mathcal{B}\rightarrow\diamond\mathcal{W}}$ works for any asynchronous system that has a majority of correct processes. In other words, $\diamond\mathcal{W}$ is reducible to $\mathcal{B}$ in such a system.

We prove that $\diamond\mathcal{W}$ is reducible to *any* failure detector $\mathcal{D}$ that can be used to solve Consensus (this result holds for any asynchronous system). We show this reduction by giving a distributed algorithm $T_{\mathcal{D}\rightarrow\diamond\mathcal{W}}$ that transforms any such $\mathcal{D}$ into $\diamond\mathcal{W}$. Thus, in a precise sense, the failure detector $\diamond\mathcal{W}$ is necessary and sufficient for solving Consensus in asynchronous systems (with a majority of correct processes). This result is further evidence to the importance of $\diamond\mathcal{W}$ for fault-tolerance in asynchronous distributed computing.

In our discussion so far, we focused on the Consensus problem. In Section 3.6, we show that Consensus is equivalent to Atomic Broadcast in asynchronous systems with crash failures. This is shown by reducing each problem to the other.[6] In other words, a solution for one automatically yields a solution for the other. Both reductions apply to any asynchronous system (in particular, they do *not* require the assumption of a failure detector). Thus, Atomic Broadcast can be solved using the unreliable failure detectors described in this thesis. Furthermore, $\diamond\mathcal{W}$ is the weakest failure detector that can be used to solve Atomic Broadcast.

A different tack on circumventing the unsolvability of Consensus is pursued in [DDS87] and [DLS88]. The approach of those papers is based on the observation that between the completely synchronous and completely asynchronous models of distributed systems there lie a variety of intermediate *partially synchronous* models. In particular, those two papers consider 34 different models of partial synchrony and for each model determine whether or not Consensus can be solved. In this thesis, we argue that partial synchrony assumptions can be encapsulated in the unreliability of the failure detector. For example, we show how to implement one of our failure detectors (which is stronger than $\diamond\mathcal{W}$), in the models of partial

[6] They are actually equivalent even in asynchronous systems with *arbitrary*, i.e., "Byzantine", failures. However, that reduction is more complex and is omitted from this thesis.

synchrony considered in [DLS88]. This immediately implies that Consensus and Atomic Broadcast can be solved in these models. Thus, our approach can be used to unify several seemingly unrelated models of partial synchrony.[7]

As we argued earlier, using the asynchronous model of computation is highly desirable in many applications: it results in code that is simple, portable and robust. However, the fact that fundamental problems such as Consensus and Atomic Broadcast have no (deterministic) solutions in this model is a major obstacle to its use in fault-tolerant distributed computing. Our model of unreliable failure detectors provides a natural and simple extension of the asynchronous model of computation, in which Consensus and Atomic Broadcast can be solved deterministically. Thus, this extended model retains the advantages of asynchrony without inheriting its disadvantages. We believe that this approach is an important contribution towards bridging the gap between known theoretical impossibility results and the need for fault-tolerant solutions in real systems.

The remainder of this thesis is organised as follows. In Chapter 2, we informally describe several different models of fault-tolerant distributed computing. This puts our work into perspective.

In Chapter 3, we briefly describe our model and introduce eight failure detectors in terms of their abstract properties. We describe solutions to the Consensus problem with each one of these eight failure detectors. Finally, we show that Consensus and Atomic Broadcast are equivalent. Thus, all our solutions to the Consensus problem can be transformed into solutions to Atomic Broadcast.

In Chapter 4, we present a more detailed model of asynchronous distributed computing with unreliable failure detection. We prove that $\Diamond\mathcal{W}$ is reducible to any failure detector for solving Consensus in this model. This shows that $\Diamond\mathcal{W}$ is the weakest failure detector for solving Consensus in asynchronous systems.

Chapter 5 discusses related work, and in particular, we describe an implemen-

[7]For a more detailed discussion on this, see Chapter 5.

tation of an unreliable failure detector that is more powerful than $\diamond\mathcal{W}$, in several models of partial synchrony. In the Appendix, we define a failure detector hierarchy based on the strength of their accuracy and derive lower bounds on fault-tolerance.

Chapter 2

Models of distributed systems

Problems in fault-tolerant distributed computing have been studied in a variety of computational models. In order to put our work into perspective, we give a broad overview of some of these models. Such models fall into two broad categories, *message-passing* and *shared-memory.* In the former, processes communicate by sending and receiving messages over the links of a network; in the latter, they communicate by accessing shared objects, such as registers, queues, etc. In this thesis we focus on message-passing models. The following parameters determine a particular message-passing model:

Types of process failures: A process is *faulty* in an execution if its behaviour deviates from that prescribed by the algorithm it is running; otherwise, it is *correct.* Several types of failures have been studied in the literature. These include (1) *crash* failures—in which a faulty process stops prematurely and does nothing from that point on [LF82], (2) *omission* failures—in which a faulty process can intermittently omit to send or receive messages [Had84, PT86], and (3) *arbitrary* failures—in which faulty processes can behave arbitrarily [LSP82]. In this thesis, we only consider crash failures.

Types of communication failures: Several models of link failures have been considered in the literature, including crash, omission and arbitrary failures. In this thesis, we assume that the communication subsystem is reliable, i.e., no link failures occur.

Network topology: Several types of network topologies have been studied in the literature. These fall into three broad categories: *point-to-point* networks, *broadcast* networks, and *mixed* networks that permit broadcasts to a limited number of processes. This thesis focuses on point-to-point networks.

Several types of point-to-point networks have been studied. These include (1) completely connected networks—in which there is a link between every pair of processes, (2) rings—in which processes are arranged in a ring, etc. Though the physical network is almost never completely connected, network layer protocols build *virtual* links between every pair of processes. Thus, in this thesis, we assume that the network is completely connected.

Deterministic versus randomized processes: A process' behavior may be either *deterministic* or *randomized.* In general, a process can be modeled as a (possibly infinite state) automaton. Roughly speaking, the state transition relation of a deterministic process *uniquely* determines the state that results from the execution of each step on the current state. With a randomized process, the execution of a given step on the current state may result in one of several possible states, and each such transition has an associated probability. Informally, a process can "toss coins" to determine which transition to take.

Since the impossibility of Consensus applies to deterministic processes, they are the primary focus of this thesis. However, our results of Section 3.6, regarding the equivalence of Consensus and Atomic Broadcast, apply to both deterministic and randomized processes.

Synchrony: Synchrony is an attribute of both processes and communication. We say that a system is *synchronous* if it satisfies the following properties:

- There is a known upper bound δ on message delay; this consists of the time it takes for sending, transporting, and receiving a message over a link.
- Every process p has a local clock C_p with known bounded rate of drift $\rho \geq 0$ with respect to real-time. That is, for all p and all $t > t'$,

$$(1+\rho)^{-1} \leq \frac{C_p(t) - C_p(t')}{(t - t')} \leq (1+\rho)$$

  where $C_p(t)$ is the reading of C_p at real-time t.
- There are known upper and lower bounds on the time required by a process to execute a step.

A system is *asynchronous* if there is no bound on message delay, clock drift, or the time necessary to execute a step. Thus, to say that a system is asynchronous is to make *no* timing assumptions whatsoever.

The synchronous and asynchronous models are the two extremes of a spectrum of possible models. Many intermediate models have also been studied. For example, processes may have bounded speeds and perfectly synchronized clocks, but message delays may be unbounded [DDS87]. Or, message delays may be bounded but unknown [DLS88].

This thesis focuses on the asynchronous model of computation. However, our work has a direct impact on several models of *partial synchrony* (this is considered in more detail in Section 5.1).

Chapter 3

Solving agreement problems with unreliable failure detectors

In this chapter, we describe algorithms for Consensus and Atomic Broadcast using unreliable failure detectors. To do so, we first present an informal model of distributed computing with unreliable failure detectors. Although informal, the model that we give here is sufficient for the presentation of the algorithms and their proofs. In Chapter 4, we extend our model and make it more precise in order to prove a subtle lower bound.

3.1 The model

We consider *asynchronous* distributed systems in which there is no bound on message delay, clock drift, or the time necessary to execute a step. Our model of asynchronous computation with failure detection is patterned after the one in [FLP85]. The system consists of a set of n *processes*, $\Pi = \{p_1, p_2, \ldots, p_n\}$. Every pair of processes is connected by a reliable communication channel.

To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

3.1.1 Failures and failure patterns

Processes can fail by *crashing*, i.e., by prematurely halting. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we say p *crashes in* F and if $p \in \mathit{correct}(F)$ we say p *is correct in* F. We consider only failure patterns F such that at least one process is correct, i.e., $\mathit{correct}(F) \neq \emptyset$.

3.1.2 Failure detectors

Each failure detector module outputs the set of processes that it currently suspects to have crashed.[1] A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}. $H(p,t)$ is the value of the failure detector module of process p at time t. If $q \in H(p,t)$, we say that p *suspects* q *at time* t *in* H. We omit references to H when it is obvious from the context. Note that the failure detector modules of two different processes need not agree on the list of processes that are suspected to have crashed, i.e., if $p \neq q$ then $H(p,t) \neq H(q,t)$ is possible.

Informally, a failure detector $\mathcal{D}$ provides (possibly incorrect) information about the failure pattern F that occurs in an execution. Formally, *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories $\mathcal{D}(F)$. This is the set of all failure detector histories that could occur in executions with failure pattern F and failure detector $\mathcal{D}$.[2]

In this thesis, we do not define failure detectors in terms of specific *implementations*. Such implementations would have to refer to low-level network parameters, such as the network topology, the message delays, and the accuracy of the local

[1] In Chapter 4 we study a more general class of failure detectors: their modules can output values from an arbitrary range.

[2] In general, there are many executions with the same failure pattern F (e.g, these executions may differ by the pattern of their message exchange). For each such execution, $\mathcal{D}$ may give a different failure detector history.

clocks. To avoid this problem, we specify a failure detector in terms of two *abstract properties* that it must satisfy: *completeness* and *accuracy*. This allows us to design applications and prove their correctness relying solely on these properties.

3.1.3 Completeness

We consider two completeness properties:

- *Strong completeness*: Eventually every process that crashes is permanently suspected by *every* correct process. Formally, $\mathcal{D}$ satisfies strong completeness if:

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

- *Weak completeness*: Eventually every process that crashes is permanently suspected by *some* correct process. Formally, $\mathcal{D}$ satisfies weak completeness if:

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

However, completeness by itself is not a useful property. To see this, consider a failure detector which causes every process to permanently suspect every other process in the system. Such a failure detector trivially satisfies strong completeness but is clearly useless since it provides no information about failures. To be useful, a failure detector must also satisfy some accuracy property that restricts the *mistakes* that it can make. We now consider such properties.

3.1.4 Accuracy

Consider the following two accuracy properties:

- *Strong accuracy*: No process is suspected before it crashes. Formally, $\mathcal{D}$ satisfies strong accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in \Pi - F(t) : p \notin H(q,t)$$

Since it is difficult (if not impossible) to achieve strong accuracy in many practical systems, we also define:

- *Weak accuracy*: Some correct process is never suspected. Formally, $\mathcal{D}$ satisfies weak accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathcal{T}, \forall q \in \Pi - F(t) : p \notin H(q,t)$$

Even weak accuracy guarantees that at least one correct process is *never* suspected. Since this type of accuracy may be difficult to achieve, we consider failure detectors that may suspect *every* process at one time or another. Informally, we only require that strong accuracy or weak accuracy are *eventually* satisfied. The resulting properties are called *eventual strong accuracy* and *eventual weak accuracy*, respectively.

For example, eventual strong accuracy requires that there is a time after which strong accuracy holds. Formally, $\mathcal{D}$ satisfies eventual strong accuracy if:

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

An observation is now in order. Since all faulty processes will crash after some finite time, we have:

$$\forall F, \exists t \in \mathcal{T}, \forall t' \geq t : \Pi - F(t') = \mathit{correct}(F)$$

Thus, an equivalent and simpler formulation of eventual strong accuracy is:

- *Eventual strong accuracy*: There is a time after which correct processes are not suspected by any correct process. Formally, $\mathcal{D}$ satisfies eventual strong accuracy if:

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

Similarly, we specify eventual weak accuracy as follows:

- *Eventual weak accuracy*: There is a time after which some correct process is never suspected by any correct process. Formally, $\mathcal{D}$ satisfies eventual weak accuracy if:

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

We will refer to eventual strong accuracy and eventual weak accuracy as *eventual accuracy* properties, and strong accuracy and weak accuracy as *perpetual accuracy* properties.

3.1.5 Some failure detector definitions

A failure detector can be specified by stating the completeness property and the accuracy property that it must satisfy. Combining the two completeness properties with the four accuracy properties that we defined in the previous section gives rise to the eight different failure detectors defined in Figure 3.1. For example, we say that a failure detector is *Eventually Strong* if it satisfies strong completeness and eventual weak accuracy. We denote such a failure detector by $\diamond\mathcal{S}$.

3.1.6 Algorithms and runs

In this thesis, we focus on algorithms that use unreliable failure detectors. To describe such algorithms, we only need informal definitions of algorithms and runs. More formal definitions are given in Chapter 4.

An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in *steps* of A. In each step, a process (1) may receive a message that was sent to it, (2) queries its failure detector module, (3) undergoes a state transition, and (4) may send a message. Since we model asynchronous systems, messages may experience arbitrary (but finite) delays. Furthermore, there is no bound on relative process speeds.

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

Figure 3.1: Some failure detector specifications based on accuracy and completeness.

Informally, a *run of algorithm 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 history of failure detector $\mathcal{D}$ for failure pattern F, I is an initial configuration of A, S is an infinite sequence of steps of A, and T is a list of increasing time values indicating when each step in S occurred. A run must satisfy certain well-formedness and fairness properties. In particular, (1) a process cannot take a step after it crashes, (2) when a process takes a step and queries its failure detector module, it gets the current value output by its local failure detector module, and (3) every process that is correct in F takes an infinite number of steps in S and eventually receives every message sent to it.

We use the following notation. Let v be a variable in algorithm A. We denote by v_p process p's copy of v. The history of v in run R is denoted by v^R, i.e., $v^R(p,t)$ is the value of v_p at time t in run R. We denote by $\mathcal{D}_p$ process p's local failure detector module. Thus, the value of $\mathcal{D}_p$ at time t in run $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ is $H_{\mathcal{D}}(p,t)$.

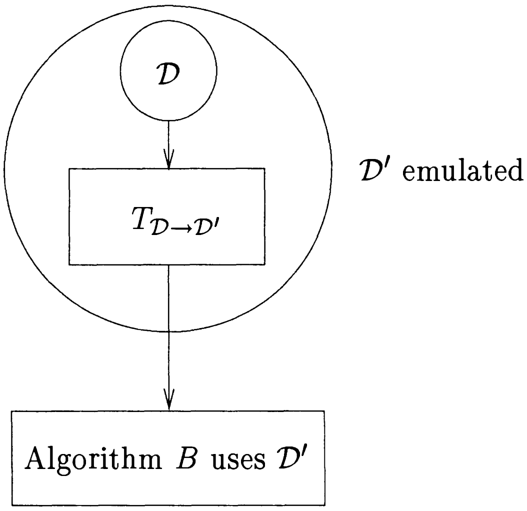

Figure 3.2: Transforming $\mathcal{D}$ into $\mathcal{D}'$

3.1.7 Reducibility

We now define what it means for an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ to transform a failure detector $\mathcal{D}$ into another failure detector $\mathcal{D}'$. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ to maintain a variable $output_p$ at every process p. This variable, reflected in the local state of p, emulates the output of $\mathcal{D}'$ at p. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ if and only if for every run $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, $output^R \in \mathcal{D}'(F)$.

Given the reduction algorithm $T_{\mathcal{D}\to\mathcal{D}'}$, anything that can be done using failure detector $\mathcal{D}'$, can be done using $\mathcal{D}$ instead. To see this, suppose a given algorithm B requires failure detector $\mathcal{D}'$, but only $\mathcal{D}$ is available. We can still execute B as follows. Concurrently with B, processes run $T_{\mathcal{D}\to\mathcal{D}'}$ to transform $\mathcal{D}$ into $\mathcal{D}'$. We modify Algorithm B at process p as follows: whenever p is required to query its failure detector module, p reads the current value of $output_p$ (which is concurrently maintained by $T_{\mathcal{D}\to\mathcal{D}'}$) instead. This is illustrated in Figure 3.2.

Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ provides at least as much information about process failures as $\mathcal{D}'$ does. Thus, if there is an algorithm

$T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ into $\mathcal{D}'$, we write $\mathcal{D} \succeq \mathcal{D}'$ and say that $\mathcal{D}'$ *is reducible to* $\mathcal{D}$; we also say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$. If $\mathcal{D} \succeq \mathcal{D}'$ and $\mathcal{D}' \succeq \mathcal{D}$, we write $\mathcal{D} \cong \mathcal{D}'$ and say that $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent*.

Note that, in general, $T_{\mathcal{D}\to\mathcal{D}'}$ need not emulate *all* the failure detector histories of $\mathcal{D}'$; what we do require is that all the failure detector histories it emulates be histories of $\mathcal{D}'$.

Consider the "identity" transformation $T_{\mathcal{D}\to\mathcal{D}}$ in which each process p periodically writes the current value output by its local failure detector module into $output_p$. The following is immediate from $T_{\mathcal{D}\to\mathcal{D}}$ and the definition of reducibility.

Observation 1: $\mathcal{P} \succeq \mathcal{Q}$, $\mathcal{S} \succeq \mathcal{W}$, $\Diamond\mathcal{P} \succeq \Diamond\mathcal{Q}$, $\Diamond\mathcal{S} \succeq \Diamond\mathcal{W}$.

3.2 From weak completeness to strong completeness

In Figure 3.3, we give a reduction algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms any given failure detector $\mathcal{D}$ that satisfies weak completeness, into a failure detector $\mathcal{D}'$ that satisfies strong completeness. Furthermore, for each failure detector $\mathcal{D}$ defined in Figure 3.1, $T_{\mathcal{D}\to\mathcal{D}'}$ gives a failure detector $\mathcal{D}'$ that has the same accuracy property as $\mathcal{D}$. Roughly speaking, $T_{\mathcal{D}\to\mathcal{D}'}$ strengthens the completeness property while preserving accuracy.

This result allows us to focus on the failure detectors that are defined in the first row of Figure 3.1, i.e., those with strong completeness. This is because, $T_{\mathcal{D}\to\mathcal{D}'}$ (together with Observation 1) shows that every failure detector in the second row of Figure 3.1 is actually *equivalent* to the corresponding failure detector above it in that figure.

Informally, $T_{\mathcal{D}\to\mathcal{D}'}$ works as follows. Every process p periodically sends $(p, suspects_p)$—where $suspects_p$ denotes the set of processes that p suspects according to its local failure detector module—to all the processes. When a process

Every process p executes the following:

$output_p \leftarrow \emptyset$

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

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

Figure 3.3: $T_{\mathcal{D}\rightarrow\mathcal{D}'}$: From Weak Completeness to Strong Completeness

q receives a message of the form $(p, suspects_p)$, it adds $suspects_p$ to $output_q$ and removes p from $output_q$.

Let $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ be an arbitrary run of $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ using failure detector $\mathcal{D}$. In the following, the run R and its failure pattern F are fixed. Thus, when we say that a process crashes we mean that it crashes in F. Similarly, when we say that a process is correct, we mean that it is correct in F. We will show that $output^R$ satisfies the following properties:

P1 : *(Transforming weak completeness into strong completeness)* Let p be any process that crashes. If eventually *some* correct process permanently suspects p in $H_{\mathcal{D}}$, then eventually *all* correct processes permanently suspect p in $output^R$. More formally:

$$\forall p \in crashed(F):$$
$$\exists t \in \mathcal{T}, \exists q \in correct(F), \forall t' \geq t : p \in H_{\mathcal{D}}(q, t')$$
$$\Rightarrow \quad \exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \in output^R(q, t')$$

P2 : *(Preserving perpetual accuracy)* Let p be any process. If no process suspects

p in $H_{\mathcal{D}}$ before time t, then no process suspects p in $output^R$ before time t. More formally:

$$
\begin{array}{ll}
\forall p \in \Pi, \forall t \in \mathcal{T}: & \\
& \forall t' < t, \forall q \in \Pi - F(t') : p \notin H_{\mathcal{D}}(q, t') \\
\Rightarrow & \forall t' < t, \forall q \in \Pi - F(t') : p \notin output^R(q, t')
\end{array}
$$

P3 : *(Preserving eventual accuracy)* Let p be any correct process. If there is a time after which no correct process suspects p in $H_{\mathcal{D}}$, then there is a time after which no correct process suspects p in $output^R$. More formally:

$$
\begin{array}{ll}
\forall p \in correct(F): & \\
& \exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \notin H_{\mathcal{D}}(q, t') \\
\Rightarrow & \exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \notin output^R(q, t')
\end{array}
$$

Lemma 2: $T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P1.

PROOF: Let p be any process that crashes. Suppose that there is a time t after which some correct process q permanently suspects p in $H_{\mathcal{D}}$. We must show that there is a time after which every correct process suspects p in $output^R$.

Since p crashes, there is a time t' after which no process receives a message from p. Consider the execution of Task 1 by process q after time $t_p = \max(t, t')$. Process q sends a message of the type $(q, suspects_q)$ with $p \in suspects_q$ to all processes. Eventually, every correct process receives $(q, suspects_q)$ and adds p to *output* (see Task 2). Since no correct process receives any messages from p after time t' and $t_p \geq t'$, no correct process removes p from *output* after time t_p. Thus, there is a time after which every correct process permanently suspects p in $output^R$. □

Lemma 3: $T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P2.

PROOF: Let p be any process. Suppose that there is a time t before which no process suspects p in $H_{\mathcal{D}}$. No process sends a message of the type $(-, \mathit{suspects})$ with $p \in \mathit{suspects}$ before time t. Thus, no process q adds p to output_q before time t. $\square$

Lemma 4: $T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P3.

PROOF: Let p be any correct process. Suppose that there is a time t after which no correct process suspects p in $H_{\mathcal{D}}$. Thus, all processes that suspect p after time t eventually crash. Thus, there is a time t' after which no correct process receives a message of the type $(-, \mathit{suspects})$ with $p \in \mathit{suspects}$.

Let q be any correct process. We must show that there is a time after which q does not suspect p in output^R.

Consider the execution of Task 1 by process p after time t'. Process p sends a message $m = (p, \mathit{suspects}_p)$ to q. When q receives m, it removes p from output_q (see Task 2). Since q does not receive any messages of the type $(-, \mathit{suspects})$ with $p \in \mathit{suspects}$ after time t', q does not add p to output_q after time t'. Thus, there is a time after which q does not suspect p in output^R. $\square$

Theorem 5: $T_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{Q}$ into $\mathcal{P}$, $\mathcal{W}$ into $\mathcal{S}$, $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$, and $\diamond\mathcal{W}$ into $\diamond\mathcal{S}$.

PROOF: By Lemma 2, $T_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{Q}$, $\mathcal{W}$, $\diamond\mathcal{Q}$, and $\diamond\mathcal{W}$, into failure detectors that satisfy strong completeness. By Lemma 3, $T_{\mathcal{D}\to\mathcal{D}'}$ preserves the strong accuracy of $\mathcal{Q}$ and the weak accuracy of $\mathcal{W}$. By Lemma 4, $T_{\mathcal{D}\to\mathcal{D}'}$ preserves the eventual strong accuracy of $\diamond\mathcal{Q}$ and the eventual weak accuracy of $\diamond\mathcal{W}$. The theorem immediately follows. $\square$

By Theorem 5 and Observation 1, we have:

Corollary 6: $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$, and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$.

Every process p executes the following:

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

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

Figure 3.4: Reliable Broadcast by message diffusion

3.3 Reliable Broadcast

We now define Reliable Broadcast, a communication primitive that we often use in our algorithms. Informally, Reliable Broadcast guarantees that (1) all correct processes deliver the same set of messages, (2) all messages broadcast by correct processes are delivered, and (3) no spurious messages are ever delivered. Formally, Reliable Broadcast is defined in terms of two primitives, *R-broadcast*(m) and *R-deliver*(m) where m is a message drawn from a set of possible messages. When a process executes *R-broadcast*(m), we say that it *R-broadcasts* m, and when a process executes *R-deliver*(m), we say that it *R-delivers* m. Reliable Broadcast satisfies the following three properties:[3]

Validity: If a correct process R-broadcasts a message m, then all correct processes eventually R-deliver m.

Agreement: If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.

Uniform integrity: For any message m, each process R-delivers m at most once,

[3]For simplicity, we assume that each message is unique. In practice, this can be achieved by tagging the identity of the sender and a sequence number on each message.

and only if m was R-broadcast by some process.

In Figure 3.4, we give a simple Reliable Broadcast algorithm for asynchronous systems. Informally, when a process receives a message for the first time, it relays the message to all processes and then R-delivers it. It is easy to show that this algorithm satisfies validity, agreement and uniform integrity in asynchronous systems with up to $n - 1$ crash failures. The proof is obvious and therefore omitted.

3.4 The Consensus problem

In the Consensus problem, all correct processes propose a value and must reach a unanimous and irrevocable decision on some value that is related to the proposed values [Fis83]. We define the Consensus problem in terms of two primitives, *propose*(v) and *decide*(v), where v is a value drawn from a set of possible proposed values. When a process executes *propose*(v), we say that it *proposes* v; similarly, when a process executes *decide*(v), we say that it *decides* v. The *Consensus* problem is specified as follows:

Termination: Every correct process eventually decides some value.

Uniform validity: If a process decides v, then v was proposed by some process.[4]

Uniform integrity: Every process decides at most once.

Agreement: No two correct processes decide differently.

It has been proved that there is no deterministic algorithm for Consensus in asynchronous systems that are subject to even a single crash failure [FLP85,DDS87]. We now show how to use unreliable failure detectors to solve Consensus in asynchronous systems.

[4]The validity condition captures the relation between the decision value and the proposed values. Changing this condition results in other types of Consensus [Fis83].

3.5 Solving Consensus using unreliable failure detectors

We now show how to solve Consensus using each one of the eight failure detectors defined in Figure 3.1. By Theorem 5, we only need to show how to solve Consensus using the four failure detectors that satisfy strong completeness, namely, $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$.

Solving Consensus with the Perfect Failure Detector $\mathcal{P}$ is simple, and is left as an exercise for the reader. In Section 3.5.1, we give a Consensus algorithm that uses $\mathcal{S}$. In Section 3.5.2, we give a Consensus algorithm that uses $\diamond\mathcal{S}$. Since $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$, this algorithm also solves Consensus with $\diamond\mathcal{P}$.

The Consensus algorithm that uses $\mathcal{S}$ can tolerate any number of failures. In contrast, the one that uses $\diamond\mathcal{S}$ requires a majority of correct processes. We show that this requirement is necessary even if one uses $\diamond\mathcal{P}$, a failure detector that is stronger than $\diamond\mathcal{S}$. Thus, our algorithm for solving Consensus using $\diamond\mathcal{S}$ (or $\diamond\mathcal{P}$) is optimal with respect to the number of failures that it tolerates.

3.5.1 Using a Strong Failure Detector $\mathcal{S}$

Given any Strong Failure Detector $\mathcal{S}$, the algorithm in Figure 3.5 solves Consensus in asynchronous systems. This algorithm runs through 3 phases. In Phase 1, processes execute $n-1$ asynchronous rounds (r_p denotes the current round number of process p) during which they broadcast and relay their proposed values. Each process p waits until it receives a round r message from every process that is not in $\mathcal{S}_p$, before proceeding to round $r+1$. Note that it is possible that while p is waiting for a message from q in round r, q is added to $\mathcal{S}_p$. By the above rule, p stops waiting for q's message and proceeds to round $r+1$.

By the end of Phase 2, correct processes agree on a vector based on the proposed values of all processes. The ith element of this vector either contains the proposed value of process p_i or $\perp$. We will show that this vector contains the proposed value of at least one process. In Phase 3, correct processes decide the first non-trivial component of this vector.

Let f denote the maximum number of processes that may crash.[5] Phase 1 of the algorithm consists of $n-1$ rounds, rather than the usual $f+1$ rounds of traditional Consensus algorithms (for synchronous systems). Intuitively, this is because even a *correct* process p may be suspected to have crashed by other processes. In this case, p's messages may be ignored, and p appears to commit "send-omission" failures. Thus, up to $n-1$ processes may appear to commit such failures (rather than f). Note that because $\mathcal{S}$ satisfies weak accuracy (namely, some correct process is never suspected), the maximum number of processes that may fail or appear to fail is $n-1$ rather than n.

$V_p[q]$ denotes p's current estimate of q's proposed value. $\Delta_p[q] = v_q$ at the end of round r if and only if p receives v_q, the value proposed by q, for the first time in round r.

Let $R = \langle F, H_{\mathcal{S}}, I, S, T\rangle$ be any run of the algorithm in Figure 3.5 using $\mathcal{S}$ in which all correct processes propose a value. We have to show that termination, uniform validity, agreement and uniform integrity hold.

Lemma 8: For all p and q, and in all phases, $V_p[q]$ is either v_q or $\perp$.

PROOF: Obvious from the algorithm in Figure 3.5. □

Lemma 9: Every correct process eventually reaches Phase 3.

PROOF: [*sketch*] The only way a correct process p can be prevented from reaching Phase 3 is by blocking forever at one of the two **wait** statements (in Phase 1 and 2, respectively). This can happen only if p is waiting forever for a message from a process q and q never joins $\mathcal{S}_p$. There are two cases to consider:

1. q crashes. Since $\mathcal{S}$ satisfies strong completeness, there is a time after which

[5] In the literature, t is often used instead of f, the notation adopted here. In this thesis, we reserve t to denote real-time.

Every process p executes the following:

procedure *propose*(v_p)
  $V_p \leftarrow \langle \bot, \bot, \ldots, \bot \rangle$ {*p's estimate of the proposed values*}
  $V_p[p] \leftarrow v_p$
  $\Delta_p \leftarrow V_p$

Phase 1: {*asynchronous rounds* r_p, $1 \le r_p \le n-1$}
  for $r_p \leftarrow 1$ to $n-1$
    send (r_p, Δ_p, p) to all
    wait until $[\forall q:$ received (r_p, Δ_q, q) **or** $q \in \mathcal{S}_p]$
      {*Query the failure detector*}
    $msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$
    $\Delta_p \leftarrow \langle \bot, \bot, \ldots, \bot \rangle$
    for $k \leftarrow 1$ to n
      if $V_p[k] = \bot$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ with $\Delta_q[k] \neq \bot$ **then**
        $V_p[k] \leftarrow \Delta_q[k]$
        $\Delta_p[k] \leftarrow \Delta_q[k]$

Phase 2: send V_p to all
  wait until $[\forall q:$ received V_q **or** $q \in \mathcal{S}_p]$ {*Query the failure detector*}
  $lastmsgs_p \leftarrow \{V_q \mid \text{received } V_q\}$
  for $k \leftarrow 1$ to n
    if $\exists V_q \in lastmsgs_p$ with $V_q[k] = \bot$ **then** $V_p[k] \leftarrow \bot$

Phase 3: *decide*(first non-$\bot$ component of V_p)

Figure 3.5: Solving Consensus using $\mathcal{S}$.

$q \in \mathcal{S}_p$.

2. q does not crash. In this case, we can show (by an easy but tedious induction on the round number) that q eventually sends the message p is waiting for.

In both cases p is not blocked forever and reaches Phase 3. $\square$

Since $\mathcal{S}$ satisfies weak accuracy there is a correct process c that is never suspected by any process, i.e., $\forall t \in \mathcal{T}, \forall p \in \Pi - F(t) : c \notin H_{\mathcal{S}}(p,t)$. Let Π_1 denote the set of processes that complete all $n-1$ rounds of Phase 1, and Π_2 denote the set of processes that complete Phase 2. We say $V_p \leq V_q$ if and only if for all $k \in \Pi$, $V_p[k]$ is either $V_q[k]$ or $\perp$.

Lemma 10: In every round r, $1 \leq r \leq n-1$, all processes $p \in \Pi_1$ receive (r, Δ_c, c) from process c, i.e., (r, Δ_c, c) is in $msgs_p[r]$.

PROOF: Since $p \in \Pi_1$, p completes all $n-1$ rounds of Phase 1. At each round r, since $c \notin \mathcal{S}_p$, p waits for and receives the message (r, Δ_c, c) from c. $\square$

Lemma 11: For all $p \in \Pi_1$, $V_c \leq V_p$ at the end of Phase 1.

PROOF: Suppose for some process q, $V_c[q] \neq \perp$ at the end of Phase 1. From Lemma 8, $V_c[q] = v_q$. Consider any $p \in \Pi_1$. We must show that $V_p[q] = v_q$ at the end of Phase 1. This is obvious if $p = c$, thus we consider the case where $p \neq c$.

Let r be the first round in which c received v_q (if $c = q$, we define r to be 0). From the algorithm, it is clear that $\Delta_c[q] = v_q$ at the end of round r. There are two cases to consider:

1. $r \leq n-2$. In round $r+1 \leq n-1$, c relays v_q by sending the message $(r+1, \Delta_c, c)$ with $\Delta_c[q] = v_q$ to all. From Lemma 10, p receives $(r+1, \Delta_c, c)$ in round $r+1$. From the algorithm, it is clear that p sets $V_p[q]$ to v_q by the end of round $r+1$.

2. $r = n - 1$. In this case, c received v_q for the first time in round $n - 1$. Since each process relays v_q (in its vector Δ) at most once, it is easy to see that v_q was relayed by all $n - 1$ processes in $\Pi - \{c\}$, including p, before being received by c. Since p sets $V_p[q] = v_q$ before relaying v_q, it follows that $V_p[q] = v_q$ at the end of Phase 1. □

Lemma 12: For all $p \in \Pi_2$, $V_c = V_p$ at the end of Phase 2.

PROOF: Consider any $p \in \Pi_2$ and $q \in \Pi$. We have to show that $V_p[q] = V_c[q]$ at the end of Phase 2. There are two cases to consider:

1. $V_c[q] = v_q$ at the end of Phase 1. From Lemma 11, for all processes $p' \in \Pi_1$ (including p and c), $V_{p'}[q] = v_q$ at the end of Phase 1. Thus, for all the vectors V sent in Phase 2, $V[q] = v_q$. Hence, both $V_p[q]$ and $V_c[q]$ remain equal to v_q throughout Phase 2.

2. $V_c[q] = \perp$ at the end of Phase 1. Since $c \notin \mathcal{S}_p$, p waits for and receives V_c in Phase 2. Since $V_c[q] = \perp$, p sets $V_p[q] \leftarrow \perp$ at the end of Phase 2. □

Lemma 13: For all $p \in \Pi_2$, $V_p[c] = v_c$ at the end of Phase 2.

PROOF: It is clear from the algorithm that $V_c[c] = v_c$ at the end of Phase 1. From Lemma 11, for all $q \in \Pi_1$, $V_q[c] = v_c$ at the end of Phase 1. Thus, no process sends V with $V[c] = \perp$ in Phase 2. From the algorithm, it is clear that for all $p \in \Pi_2$, $V_p[c] = v_c$ at the end of Phase 2. □

Theorem 14: Given any Strong Failure Detector $\mathcal{S}$, the algorithm in Figure 3.5 solves Consensus in asynchronous systems with $f < n$.

PROOF: From the algorithm in Figure 3.5, it is clear that no process decides more than once, and this satisfies the uniform integrity requirement of Consensus. From Lemma 9, every correct process eventually reaches Phase 3. From Lemma 13, the vector V_p of every correct has at least one non-$\perp$ component in Phase 3 (namely,

$V_p[c] = v_c$). From the algorithm, every process p that reaches Phase 3, decides on the first non-$\perp$ component of V_p. Thus, every correct process decides some non-$\perp$ value in Phase 3—and this satisfies termination of Consensus. From Lemma 12, all processes that reach Phase 3 have the same vector V. Thus, all correct processes decide the same value, and agreement of Consensus is satisfied. From Lemma 8, this non-$\perp$ decision value is the proposed value of some process. Thus, uniform validity of Consensus is also satisfied. $\square$

By Theorems 5 and 14, we have:

Corollary 15: Given any Weak Failure Detector $\mathcal{W}$, Consensus is solvable in asynchronous systems with $f < n$.

3.5.2 Using an Eventually Strong Failure Detector $\diamond\mathcal{S}$

Our previous solution to Consensus used $\mathcal{S}$, a failure detector that satisfies weak accuracy: at least one correct process was *never* suspected. We now solve Consensus using $\diamond\mathcal{S}$, a failure detector that only satisfies eventual weak accuracy. With $\diamond\mathcal{S}$, *all* processes may be erroneously added to the lists of suspects at one time or another. However, there is a correct process and a time after which that process is not suspected to have crashed. Note that at any given time t, processes cannot use $\diamond\mathcal{S}$ to determine whether any specific process is correct, or whether some correct process will never be suspected after time t.

Given any Eventually Strong Failure Detector $\diamond\mathcal{S}$, the algorithm in Figure 3.6 solves Consensus in asynchronous systems with a majority of correct processes. We show that solving Consensus using $\diamond\mathcal{S}$ requires this majority.[6] Thus, our algorithm is optimal with respect to the number of failures that it tolerates.

The algorithm in Figure 3.6 uses the *rotating coordinator* paradigm [Rei82,

[6]In fact, we show that a majority of correct processes is required even if one uses $\diamond\mathcal{P}$, a stronger failure detector.

CM84,DLS88,BGP89,CT90]. Computation proceeds in asynchronous "rounds". We assume that all processes have a priori knowledge that during round r, the coordinator is process $c = (r \bmod n) + 1$. All messages are either to or from the "current" coordinator. Every time a process becomes a coordinator, it tries to determine a consistent decision value. If the current coordinator is correct *and* is not suspected by any surviving process, then it will succeed, and it will R-broadcast this decision value.

The algorithm in Figure 3.6 goes through three asynchronous epochs, each of which may span several asynchronous rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked:* no other decision value is possible. In the third epoch, processes decide the locked value.

Each round of this Consensus algorithm is divided into four asynchronous phases. In Phase 1, every process sends its current estimate of the decision value timestamped with the round number in which it adopted this estimate, to the current coordinator, c. In Phase 2, c gathers $n - f$ such estimates, selects one with the largest timestamp, and sends it to all the processes as their new estimate, $estimate_c$. In Phase 3, for each process p there are two possibilities:

1. p receives $estimate_c$ from c and sends an *ack* to c to indicate that it adopted $estimate_c$ as its own estimate; or

2. upon consulting its failure detector module $\Diamond\mathcal{S}_p$, p *suspects* that c crashed, and sends a *nack* to c.

In Phase 4, c waits for $n - f$ replies (*ack*s or *nack*s). If all $n - f$ replies are *ack*s, then c knows that $n - f$ processes changed their estimates to $estimate_c$, and thus $estimate_c$ is locked. Consequently, c R-broadcasts a request to decide $estimate_c$. At any time, if a process R-delivers such a request, it decides accordingly.

The proof that the algorithm in Figure 3.6 solves Consensus is as follows. Let R be any run of the algorithm in Figure 3.6 using $\Diamond\mathcal{S}$ in which all correct processes

Every process p executes the following:

```
procedure propose(v_p)
  estimate_p ← v_p                      {denotes p's estimate of the decision value}
  state_p ← undecided
  r_p ← 0                               {r_p denotes the current round number}
  ts_p ← 0                              {the round in which estimate_p was last updated, initially 0}

  {Rotate through coordinators until decision is reached}

  while state_p = undecided
    r_p ← r_p + 1
    c_p ← (r_p mod n) + 1               {c_p is the current coordinator}

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

    Phase 2: {The current coordinator gathers n − f estimates and proposes a new estimate}
      if p = c_p then
        wait until [for n − f processes q : received (q, r_p, estimate_q, ts_q) from q]
        msgs_p[r_p] ← {(q, r_p, estimate_q, ts_q) | p received (q, r_p, estimate_q, ts_q) from q}
        t ← largest ts_q such that (q, r_p, estimate_q, ts_q) ∈ msgs_p[r_p]
        estimate_p ← select one estimate_q such that (q, r_p, estimate_q, t) ∈ msgs_p[r_p]
        send (p, r_p, estimate_p) to all

    Phase 3: {All processes wait for the new estimate proposed by the current coordinator}
      wait until [received (c_p, r_p, estimate_c_p) from c_p or c_p ∈ ◇S_p]{Query the failure detector}
      if [received (c_p, r_p, estimate_c_p) from c_p] then      {p received estimate_c_p from c_p}
        estimate_p ← estimate_c_p
        ts_p ← r_p
        send (p, r_p, ack) to c_p
      else send (p, r_p, nack) to c_p                          {p suspects that c_p crashed}

    Phase 4: { The current coordinator waits for n − f replies. If these replies indicate that
               n − f processes adopted its estimate, the coordinator sends a request to decide. }
      if p = c_p then
        wait until [for n − f processes q : received (q, r_p, ack) or (q, r_p, nack)]
        if [for n − f processes q : received (q, r_p, ack)] then
          R-broadcast(p, r_p, estimate_p, decide)

{When p receives a decide message, it decides}

when R-deliver(q, r_q, estimate_q, decide)
  if state_p = undecided then
    decide(estimate_q)
    state_p ← decided
```

Figure 3.6: Solving Consensus using $\diamond\mathcal{S}$

propose a value. We have to show that termination, uniform validity, agreement and uniform integrity hold.

Lemma 17: No two processes decide differently.[7]

PROOF: If no process ever decides, the lemma is trivially true. If any process decides, it must be the case that a coordinator R-broadcast a message of the type $(-,-,-,decide)$. This coordinator must have received $n-f$ messages of the type $(-,-,ack)$ in Phase 4. Let r be the smallest round number in which $n-f$ messages of the type $(-,r,ack)$ are sent to a coordinator in Phase 3. Let c denote the coordinator of round r, i.e., $c = (r \bmod n) + 1$. Let $estimate_c$ denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geq r$, if a coordinator c' sends $estimate_{c'}$ in Phase 2 of round r', then $estimate_{c'} = estimate_c$.

The proof is by induction on the round number. The claim trivially holds for $r' = r$. Now assume that the claim holds for all r', $r \leq r' < k$. Let c_k be the coordinator of round k, i.e., $c_k = (k \bmod n) + 1$. We will show that the claim holds for $r' = k$, i.e., if c_k sends $estimate_{c_k}$ in Phase 2 of round k, then $estimate_{c_k} = estimate_c$.

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

In Phase 2 of round k, c_k executes $estimate_{c_k} \leftarrow estimate_q$ where

[7] This property, called *uniform agreement*, is stronger than the agreement requirement of Consensus which applies only to *correct* processes.

$(q, k, estimate_q, t)$ is in $msgs_{c_k}[k]$. From Figure 3.6, it is clear that q adopted $estimate_q$ as its estimate in Phase 3 of round t. Thus, the coordinator of round t sent $estimate_q$ to q in Phase 2 of round t. Since $r \leq t < k$, by the induction hypothesis, $estimate_q = estimate_c$. Thus, c_k sets $estimate_{c_k} \leftarrow estimate_c$ in Phase 2 of round k. This concludes the proof of the claim.

We now show that if a process decides a value, then it decides $estimate_c$. Suppose that some process p R-delivers $(q, r_q, estimate_q, decide)$, and thus decides $estimate_q$. Process q must have R-broadcast $(q, r_q, estimate_q, decide)$ in Phase 4 of round r_q. From Figure 3.6, q must have received $n - f$ messages of the type $(-, r_q, ack)$ in Phase 4 of round r_q. By the definition of r, $r \leq r_q$. From the above claim, $estimate_q = estimate_c$. □

Lemma 18: Every correct process eventually decides some value.

PROOF: There are two possible cases:

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

2. No correct process decides. We claim that no correct process remains blocked forever at one of the **wait** statements. The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the **wait** statements. Thus, all correct processes reach the end of Phase 1 of round r: they all send a message of the type $(-, r, estimate, -)$ to the current coordinator $c = (r \bmod n) + 1$. Therefore at least $n - f$ such messages are sent to c. There are two cases to consider:

   (a) Eventually, c receives those messages and replies by sending $(c, r, estimate_c)$. Thus, c does not block forever at the **wait** statement in Phase 2.

   (b) c crashes.

In the first case, every correct process receives $(c, r, estimate_c)$. In the second case, since $\diamond\mathcal{S}$ satisfies *strong completeness*, for every correct process p there is a time after which c is permanently suspected by p, i.e., $c \in \diamond\mathcal{S}_p$. Thus in either case, no correct process blocks at the second **wait** statement (Phase 3). So every correct process sends a message of the type $(-, r, ack)$ or $(-, r, nack)$ to c in Phase 3. Since there are $n - f$ correct processes, c cannot block at the **wait** statement of Phase 4. This shows that all correct processes complete round r—a contradiction that completes the proof of our claim.

Since $\diamond\mathcal{S}$ satisfies eventual weak accuracy, there is a correct process q and a time t such that no correct process suspects q after t. Thus, all processes that suspect q after time t eventually crash and there is a time t' after which no process sends a message of the type $(-, r, nack)$ where q is the coordinator of round r (i.e., $q = (r \bmod n) + 1$). From this and the above claim, there must be a round r such that:

(a) All correct processes reach round r after time t' (when no process suspects q).

(b) q is the coordinator of round r (i.e., $q = (r \bmod n) + 1$).

In Phase 1 of round r, all correct processes send their estimates to q. In Phase 2, q receives $n - f$ such estimates, and sends $(q, r, estimate_q)$ to all processes. In Phase 3, since q is not suspected by any correct process after time t, every correct process waits for q's estimate, eventually receives it, and replies with an *ack* to q. Furthermore, no process sends a *nack* to q (that can only happen when a process suspects q). Thus in Phase 4, q receives $n - f$ messages of the type $(-, r, ack)$ (and no messages of the type $(-, r, nack)$), and q R-broadcasts $(q, r, estimate_q, decide)$. By the validity property of Reliable Broadcast, eventually all correct processes R-deliver q's message and *decide*—a contradiction. Thus case 2 is impossible, and this concludes the proof of the lemma. □

Theorem 19: Given any Eventually Strong Failure Detector $\Diamond\mathcal{S}$, the algorithm in Figure 3.6 solves Consensus in asynchronous systems with $f < \frac{n}{2}$.

PROOF:

Termination: by Lemma 18.

Agreement: by Lemma 17.

Uniform integrity: It is clear from the algorithm that no process decides more than once.

Uniform validity: from the algorithm, it is clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, uniform validity is satisfied. □

By Theorems 5 and 19, we have:

Corollary 20: Given any Eventually Weak Failure Detector $\Diamond\mathcal{W}$, Consensus is solvable in asynchronous systems with $f < \frac{n}{2}$.

Thus, the weakest failure detector considered in this thesis, $\Diamond\mathcal{W}$, is sufficient to solve Consensus in asynchronous systems. This leads to the following question: What is the weakest failure detector for solving Consensus? Using the concept of reducibility, in Chapter 4 we show that $\Diamond\mathcal{W}$ is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes. More precisely, we show:

Theorem 21: If a failure detector $\mathcal{D}$ can be used to solve Consensus in an asynchronous system, then $\mathcal{D} \succeq \Diamond\mathcal{W}$ in that system.

By Corollary 20 and Theorem 21, we have:

Corollary 22: $\Diamond\mathcal{W}$ is the weakest failure detector for solving Consensus in an asynchronous system with $f < \frac{n}{2}$.

3.5.3 A lower bound on fault-tolerance

In Section 3.5.1, we showed that failure detectors with *perpetual* accuracy (i.e., $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, or $\mathcal{W}$) can be used to solve Consensus in asynchronous systems with *any* number of failures. In contrast, with failure detectors with *eventual* accuracy (i.e., $\Diamond\mathcal{P}$, $\Diamond\mathcal{Q}$, $\Diamond\mathcal{S}$, or $\Diamond\mathcal{W}$), our Consensus algorithms required a majority of the processes to be correct. We now show that this requirement is necessary: Any algorithm that uses $\Diamond\mathcal{P}$ (the strongest of our four failure detectors with eventual accuracy) to solve Consensus requires a majority of correct processes. Thus, the algorithm in Figure 3.6 is optimal with respect to fault-tolerance.

Theorem 23: There is an Eventually Perfect Failure Detector $\Diamond\mathcal{P}$ such that there is no algorithm A which solves Consensus using $\Diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil\frac{n}{2}\rceil$.

Proof: We now describe the behaviour of an Eventually Perfect Failure Detector $\Diamond\mathcal{P}$ such that with every algorithm A, there is a run R_A of A using $\Diamond\mathcal{P}$ that does not satisfy the specification of Consensus. Partition the processes into two sets Π_0 and Π_1 such that Π_0 contains $\lceil\frac{n}{2}\rceil$ processes, and Π_1 contains the remaining $\lfloor\frac{n}{2}\rfloor$ processes. Consider any Consensus algorithm A, and the following two runs of A using $\Diamond\mathcal{P}$:

- Run $R_0 = \langle F_0, H_0, I, S_0, T_0 \rangle$: All processes in Π_0 propose 0, and all processes in Π_1 propose 1. All processes in Π_0 are correct in F_0, while those in Π_1 crash in F_0 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_0(t) = \Pi_1$ (this is possible since $f \geq \lceil\frac{n}{2}\rceil$). Every process in Π_0 permanently suspects every process in

Π_1, i.e., $\forall t \in \mathcal{T}$, $\forall p \in \Pi_0 : H_0(p,t) = \Pi_1$. In this run, it is clear that $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect Failure Detector.

- Run $R_1 = \langle F_1, H_1, I, S_1, T_1 \rangle$: As in R_0, all processes in Π_0 propose 0, and all processes in Π_1 propose 1. All processes in Π_1 are correct in F_1, while those in Π_0 crash in F_1 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_1(t) = \Pi_0$. Every process in Π_1 permanently suspects every process in Π_0, i.e., $\forall t \in \mathcal{T}$, $\forall p \in \Pi_1 : H_1(p,t) = \Pi_0$. Clearly, $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect Failure Detector in this run.

Assume, without loss of generality, that both R_0 and R_1 satisfy the specifications of Consensus. Let $q_0 \in \Pi_0$, $q_1 \in \Pi_1$, t_0 be the time at which q_0 decides in R_0, and t_1 be the time at which q_1 decides in R_1. There are three possible cases—in each case we construct a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of algorithm A using $\diamond\mathcal{P}$ such that $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect Failure Detector, but R_A violates the specification of Consensus.

1. In R_0, q_0 decides 1. Let $R_A = \langle F_0, H_0, I_A, S_0, T_0 \rangle$ be a run identical to R_0 except that all processes in Π_1 propose 0. Since in F_0 the processes in Π_1 crash right from the beginning of the run, R_0 and R_A are indistinguishable to q_0. Thus, q_0 decides 1 in R_A (as it did in R_0), thereby violating the uniform validity condition of Consensus.

2. In R_1, q_1 decides 0. This case is symmetric to Case 1.

3. In R_0, q_0 decides 0, and in R_1, q_1 decides 1. Construct $R_A = \langle F_A, H_A, I, S_A, T_A \rangle$ as follows. No processes crash in F_A, i.e., $\forall t \in \mathcal{T} : F_A(t) = \emptyset$. As before, all processes in Π_0 propose 0 and all processes in Π_1 propose 1. All messages from processes in Π_0 to those in Π_1 and vice-versa, are delayed until time $\max(t_0, t_1)$. Until time $\max(t_0, t_1)$, every process in Π_0 suspects every process in Π_1, and every process in Π_1 suspects every process in Π_0. After time $\max(t_0, t_1)$, no process suspects any other process, i.e.:

$$\forall t \leq \max(t_0, t_1):$$
$$\forall p \in \Pi_0 : H_A(p,t) = \Pi_1$$
$$\forall p \in \Pi_1 : H_A(p,t) = \Pi_0$$
$$\forall t > \max(t_0, t_1), \forall p \in \Pi : H_A(p,t) = \emptyset$$

Clearly, $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect Failure Detector.

Until time $\max(t_0, t_1)$, R_A is indistinguishable from R_0 for processes in Π_0, and R_A is indistinguishable from R_1 for processes in Π_1. Thus in run R_A, q_0 decides 0 at time t_0, while q_1 decides 1 at time t_1. So q_0 and q_1 decide differently in R_A, and this violates the agreement condition of Consensus. □

In the Appendix, we refine the result of Theorem 23, by considering an infinite hierarchy of failure detectors ordered by the number of mistakes they can make, and showing exactly where in this hierarchy the majority requirement becomes necessary for solving Consensus (this hierarchy contains all eight failure detectors that we defined in Figure 3.1). Note that Theorem 23 is also a corollary of Theorem 4.3 in [DLS88] together with Theorem 66.

3.6 On Atomic Broadcast

We now consider Atomic Broadcast, another fundamental problem in fault tolerant distributed computing, and show that our results on Consensus also apply to Atomic Broadcast. Informally, Atomic Broadcast requires that all correct processes deliver the same messages in the same order. Formally, *Atomic Broadcast* is a Reliable Broadcast that satisfies:

- *Total order*: If two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Total order and agreement ensure that all correct processes deliver the same *sequence* of messages. Atomic Broadcast is a powerful communication paradigm for fault-tolerant distributed computing [CM84,CASD85,BJ87,PGM89,BGT90, GSTC90,Sch90]. We now show that Consensus and Atomic Broadcast are *equivalent* in asynchronous systems with crash failures. This is shown by reducing each to the other.[8] In other words, a solution for one automatically yields a solution for the other. Both reductions apply to any asynchronous system (in particular, they do *not* require the assumption of a failure detector). This equivalence has important consequences regarding the solvability of Atomic Broadcast in *asynchronous* systems:

1. Atomic Broadcast can*not* be solved with a deterministic algorithm in asynchronous systems, even if we assume that at most one process may fail, and it can only fail by crashing. This is because Consensus has no deterministic solution in such systems [FLP85].

2. Atomic Broadcast can be solved using *randomization* or *unreliable failure detectors* in asynchronous systems. This is because Consensus is solvable with these techniques in such systems (for a survey of randomized Consensus algorithms, see [CD89]).

Consensus can be easily reduced to Atomic Broadcast as follows. To propose a value, a process atomically broadcasts it. To decide a value, a process picks the value of the first message that it atomically delivers.[9] By total order of Atomic Broadcast, all correct processes deliver the same first message. Hence they choose the same value and agreement of Consensus is satisfied. The other properties of Consensus are also easy to verify. In the next section, we reduce Atomic Broadcast to Consensus.

[8] They are actually equivalent even in asynchronous systems with arbitrary failures. However, the reduction is more complex and is omitted here.

[9] Note that this reduction does *not* require the assumption of a failure detector.

3.6.1 Reducing Atomic Broadcast to Consensus

In Figure 3.7, we show how to transform any Consensus algorithm into an Atomic Broadcast algorithm in asynchronous systems. The resulting Atomic Broadcast algorithm tolerates as many faulty processes as the given Consensus algorithm.

The reduction uses Reliable Broadcast, and repeated (possibly concurrent, but completely *independent*) executions of Consensus. Processes disambiguate between these executions by tagging all the messages pertaining to the k^{th} execution of Consensus with the number k. Tagging each message with such a number constitutes a minor modification to any given Consensus algorithm. Informally, the k^{th} execution of Consensus is used to decide on the k^{th} batch of messages to be atomically delivered. The propose and decide primitives corresponding to the k^{th} execution of Consensus are denoted by *propose*$(k, -)$ and *decide*$(k, -)$.

Our Atomic Broadcast algorithm uses the *R-broadcast*(m) and *R-deliver*(m) primitives of Reliable Broadcast. To avoid possible ambiguities between Atomic Broadcast and Reliable Broadcast, we say that a process *A-broadcasts* or *A-delivers* to refer to a broadcast or a delivery associated with Atomic Broadcast; and *R-broadcasts* or *R-delivers* to refer to a broadcast or delivery associated with Reliable Broadcast.

When a process intends to A-broadcast a message m, it *R-broadcasts* m (in Task 1). When a process p R-delivers m, it adds m to the set $R\_delivered_p$ (Task 2). Thus, $R\_delivered_p$ contains all the messages submitted for Atomic Broadcast (since the beginning) that p is currently aware of. When p A-delivers a message m, it adds m to the set $A\_delivered_p$ (in Task 3). Thus, $R\_delivered_p - A\_delivered_p$ is the set of messages that were submitted for Atomic Broadcast but not yet A-delivered by p. This set is denoted by $A\_undelivered_p$. When $A\_undelivered_p$ is not empty, p proposes $A\_undelivered_p$ as the next batch of messages to be A-delivered. $batch_p(k)$ denotes the k^{th} batch of messages that p A-delivers: it is $msgSet_p$, the set of messages agreed upon by the k^{th} execution of Consensus, *Every process p executes the following*:

Initialization:

$R\_delivered \leftarrow \emptyset$
$A\_delivered \leftarrow \emptyset$
$k \leftarrow 0$

To execute *A-broadcast*(m): { *Task 1* }

R-broadcast(m)

A-deliver(m) occurs as follows:

when *R-deliver*(m) { *Task 2* }
$R\_delivered \leftarrow R\_delivered \cup \{m\}$

when $R\_delivered - A\_delivered \neq \emptyset$ { *Task 3* }
$k \leftarrow k + 1$
$A\_undelivered \leftarrow R\_delivered - A\_delivered$
propose($k, A\_undelivered$)
wait until *decide*($k, msgSet$)
$batch(k) \leftarrow msgSet - A\_delivered$
atomically deliver all messages in $batch(k)$ in some deterministic order
$A\_delivered \leftarrow A\_delivered \cup batch(k)$

Figure 3.7: Using Consensus to solve Atomic Broadcast

minus $A\_delivered_p$, those messages that p has already A-delivered.[10]

Lemma 24: For any two correct processes p and q, and any message m, if $m \in R\_delivered_p$ then eventually $m \in R\_delivered_q$.

PROOF: If $m \in R\_delivered_p$ then p R-delivered m (in Task 2). Since p is correct, by agreement of Reliable Broadcast q eventually R-delivers m, and inserts m into $R\_delivered_q$. □

[10] It is possible for a process p to A-deliver a message m before it R-delivers m. This occurs if m was proposed by another process, and agreed upon by Consensus, before p R-delivers m.

Lemma 25: For any two correct processes p and q, and all $k \geq 1$:

1. If p executes *propose*$(k, -)$, then q eventually executes *propose*$(k, -)$.

2. If p A-delivers messages in $batch_p(k)$, then q eventually A-delivers messages in $batch_q(k)$, and $batch_p(k) = batch_q(k)$.

PROOF: The proof is by simultaneous induction on (1) and (2). For $k = 1$, we first show that if p executes *propose*$(1, -)$, then q eventually executes *propose*$(1, -)$. When p executes *propose*$(1, -)$, $R\_delivered_p$ must contain some message m. By Lemma 24, m is eventually in $R\_delivered_q$. Since $A\_delivered_q$ is initially empty, eventually $R\_delivered_q - A\_delivered_q \neq \emptyset$. Thus, q eventually executes Task 3 and *propose*$(1, -)$.

We now show that if p A-delivers messages in $batch_p(1)$, then q eventually A-delivers messages in $batch_q(1)$, and $batch_p(1) = batch_q(1)$. From the algorithm, if p A-delivers messages in $batch_p(1)$, it previously executed *propose*$(1, -)$. From part (1) of the lemma, all correct processes eventually execute *propose*$(1, -)$. By termination and uniform integrity of Consensus, every correct process eventually executes *decide*$(1, -)$ and it does so exactly once. By agreement of Consensus, all correct processes eventually execute *decide*$(1, msgSet)$ with the same $msgSet$. Since $A\_delivered_p$ and $A\_delivered_q$ are initially empty, $batch_p(1) = batch_q(1) = msgSet_p = msgSet_q$.

Now assume that the lemma holds for all k, $1 \leq k < l$. We first show that if p executes *propose*$(l, -)$, then q eventually executes *propose*$(l, -)$. When p executes *propose*$(l, -)$, $R\_delivered_p$ must contain some message m that is not in $A\_delivered_p$. Thus, m is not in $\bigcup_{k=1}^{l-1} batch_p(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. So m is not in $\bigcup_{k=1}^{l-1} batch_q(k)$. Since m is in $R\_delivered_p$, by Lemma 24, m is eventually in $R\_delivered_q$. Thus, there is a time after q A-delivers $batch_q(l-1)$ such that there is a message in $R\_delivered_q - A\_delivered_q$. So q eventually executes Task 3 and *propose*$(l, -)$.

We now show that if p A-delivers messages in $batch_p(l)$, then q A-delivers messages in $batch_q(l)$, and $batch_p(l) = batch_q(l)$. Since p A-delivers messages in $batch_p(l)$, it must have executed *propose*$(l, -)$. By part (1) of this lemma, all correct processes eventually execute *propose*$(l, -)$. By termination and uniform integrity of Consensus, every correct process eventually executes *decide*$(l, -)$ and it does so exactly once. By agreement of Consensus, all correct processes eventually execute *decide*$(l, msgSet)$ with the same $msgSet$. Note that $batch_p(l) = msgSet - \bigcup_{k=1}^{l-1} batch_p(k)$, and $batch_q(l) = msgSet - \bigcup_{k=1}^{l-1} batch_q(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. Thus, $batch_p(l) = batch_q(l)$. □

Lemma 26: The algorithm in Figure 3.7 satisfies agreement and total order.

PROOF: Immediate from Lemma 25, and the fact that correct processes A-deliver messages in each batch in the same deterministic order. □

Lemma 27: (Validity) If a correct process A-broadcasts m, then all correct processes eventually A-deliver m.

PROOF: The proof is by contradiction. Suppose some correct process p A-broadcasts m, and some correct process never A-delivers m. By Lemma 26, no correct process A-delivers m.

By Task 1 of Figure 3.7, p R-broadcasts m. By validity of Reliable Broadcast, every correct process q eventually R-delivers m, and inserts m in $R\_delivered_q$ (Task 2). Since correct processes never A-deliver m, they never insert m in $A\_delivered$. Thus, for every correct process q, there is a time after which m is *permanently* in $R\_delivered_q - A\_delivered_q$. From Figure 3.7 and Lemma 25, there is a k_1, such that for all $l > k_1$, all correct processes execute *propose*$(l, -)$, and they do so with sets that always include m.

Since all faulty processes eventually crash, there is a k_2 such that no faulty process executes *propose*$(l, -)$ with $l > k_2$. Let $k = max(k_1, k_2)$. Since all correct processes execute *propose*$(k, -)$, by termination and agreement of Consensus, all correct processes execute *decide*$(k, msgSet)$ with the same $msgSet$. By uniform validity of Consensus, some process q executed *propose*$(k, msgSet)$. From our definition of k, q is correct and $msgSet$ contains m. Thus all correct processes A-deliver m—a contradiction that concludes the proof. □

Lemma 28: (Uniform integrity) For any message m, each process A-delivers m at most once, and only if m was A-broadcast by some process.

PROOF: Suppose a process p A-delivers m. After p A-delivers m, it inserts m in $A\_delivered_p$. From the algorithm, it is clear that p cannot A-deliver m again.

From the algorithm, p executed *decide*$(k, msgSet)$ for some k and some $msgSet$ that contains m. By uniform validity of Consensus, some process q must have executed *propose*$(k, msgSet)$. So q previously R-delivered all the messages in $msgSet$, including m. By uniform integrity of Reliable Broadcast, some process r R-broadcast m. So, r A-broadcast m. □

Theorem 29: Consider any system (synchronous or asynchronous) subject to crash failures and where Reliable Broadcast can be implemented. The algorithm in Figure 3.7 transforms any algorithm for Consensus into an Atomic Broadcast algorithm.

PROOF: Immediate from Lemmata 26, 27, and 28. □

Since Reliable Broadcast can be implemented in asynchronous systems with crash failures (Section 3.3), the above theorem shows that Atomic Broadcast is reducible to Consensus in those systems. As we argued earlier, the converse is also true. Thus:

Corollary 30: Consensus and Atomic Broadcast are equivalent in asynchronous systems with crash failures.

The equivalence of Consensus and Atomic Broadcast in asynchronous systems immediately implies that our results regarding Consensus (in particular Corollaries 15 and 22, and Theorem 23) also hold for Atomic Broadcast:

Corollary 31: Given any Weak Failure Detector $\mathcal{W}$, Atomic Broadcast is solvable in asynchronous systems with $f < n$.

Corollary 32: $\diamond\mathcal{W}$ is the weakest failure detector for solving Atomic Broadcast in an asynchronous system with $f < \frac{n}{2}$.

Corollary 33: There is an Eventually Perfect Failure Detector $\diamond\mathcal{P}$ such that there is no algorithm A which solves Atomic Broadcast using $\diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil\frac{n}{2}\rceil$.

Furthermore, Theorem 29 shows that by "plugging in" any *randomized* Consensus algorithm (such as the ones in [CD89]) into the algorithm of Figure 3.7, we automatically get a randomized algorithm for Atomic Broadcast in asynchronous systems.

Corollary 34: Atomic Broadcast can be solved by randomized algorithms in asynchronous systems with $f < \frac{n}{2}$ crash failures.

Chapter 4

The weakest failure detector for solving Consensus

In the previous chapter, we showed that Consensus can be solved in asynchronous systems using unreliable failure detectors. In particular, we showed how to solve Consensus using any Eventually Weak Failure Detector. Recall that $\mathcal{D}$ is an Eventually Weak Failure Detector if, for every failure pattern F, every failure detector history $H \in \mathcal{D}(F)$ satisfies the following two properties:

- *Weak completeness*: There is a time after which every process that crashes in F is permanently suspected in H by some process that is correct in F.
- *Eventual weak accuracy*: There is a time after which some process that is correct in F is never suspected in H by any process that is correct in F.

In the previous chapter, $\Diamond\mathcal{W}$ denoted *any* Eventually Weak Failure Detector. In this chapter, we reserve $\Diamond\mathcal{W}$ to denote a particular Eventually Weak Failure Detector: For any failure pattern F, $\Diamond\mathcal{W}(F)$ consists of *all* failure detector histories that satisfy the two properties above. By definition, if $\mathcal{D}$ is any Eventually Weak Failure Detector, then for any failure pattern F, $\mathcal{D}(F) \subseteq \Diamond\mathcal{W}(F)$. Hence, $\mathcal{D} \succeq \Diamond\mathcal{W}$ (by the identity transformation). This shows that $\Diamond\mathcal{W}$ is the *weakest* among all

Eventually Weak Failure Detectors.

From Theorem 19, the failure detection properties of $\diamond\mathcal{W}$ are *sufficient* to solve Consensus in asynchronous systems (in which a majority of the processes are correct). In this chapter we show that they are they *necessary*: We prove that $\diamond\mathcal{W}$ is reducible to *any* failure detector $\mathcal{D}$ that can be used to solve Consensus (this result holds for any asynchronous system). We show this reduction by giving a distributed algorithm $T_{\mathcal{D}\rightarrow\diamond\mathcal{W}}$ that transforms any such $\mathcal{D}$ into $\diamond\mathcal{W}$. Therefore, $\diamond\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in asynchronous systems with $n > 2f$. Furthermore, if $n \leq 2f$, any failure detector that can be used to solve Consensus must be strictly stronger than $\diamond\mathcal{W}$.

The task of transforming any given failure detector $\mathcal{D}$ (that can be used to solve Consensus) into $\diamond\mathcal{W}$ runs into a serious technical difficulty for the following reasons:

- To strengthen our result, we do not restrict the output of $\mathcal{D}$ to lists of suspects. Instead, this output can be *any value* that encodes some information about failures. For example, a failure detector $\mathcal{D}$ should be allowed to output any boolean formula, such as "(not p) and (q or r)" (i.e., p is up and either q or r has crashed)—or any *encoding* of such a formula. Indeed, the output of $\mathcal{D}$ could be an arbitrarily complex (and unknown) encoding of failure information. Our transformation from $\mathcal{D}$ into $\diamond\mathcal{W}$ must be able to decode this information.

- Even if the failure information provided by $\mathcal{D}$ is not encoded, it is not clear how to extract from it the failure detection properties of $\diamond\mathcal{W}$. Consequently, if $\mathcal{D}$ is given in isolation, the task of transforming it into $\diamond\mathcal{W}$ may not be possible.

Fortunately, since $\mathcal{D}$ can be used to solve Consensus, there is a corresponding algorithm, *Consensus*$_{\mathcal{D}}$, that is somehow able to "decode" the information about failures provided by $\mathcal{D}$, and knows how to use it to solve Consensus. Our reduction algorithm, $T_{\mathcal{D}\to\diamond\mathcal{W}}$ uses *Consensus*$_{\mathcal{D}}$ to extract this information from $\mathcal{D}$ and transforms it into the properties of $\diamond\mathcal{W}$.

Since this chapter focuses on the proof of a subtle lower bound, we need to extend our model of distributed computation of Section 3.1 and make it more precise. This extended model, presented in the next section, introduces the following new concepts:

- *The range of values output by a failure detector:* As mentioned earlier, the main result of this chapter applies to failure detectors with arbitrary sets of output values. We formalise this by associating each failure detector with a range of output values.

- *The environment:* In the previous chapter, we proved that any algorithm that uses $\diamond\mathcal{W}$ to solve Consensus requires $n > 2f$. With other failure detectors the requirements may be different. For example, there is a failure detector that can be used to solve Consensus only if p_1 and p_2 do not both crash. In general whether a given failure detector can be used to solve Consensus depends upon assumptions about the underlying "environment". An environment (of an asynchronous system) is set of possible failure patterns. We will show that if $\mathcal{D}$ can be used to solve Consensus in some environment $\mathcal{E}$, then there is a transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{W}}$ that transforms $\mathcal{D}$ into $\diamond\mathcal{W}$ in $\mathcal{E}$.

4.1 The model

4.1.1 Failure detectors

Associated with each failure detector is a range $\mathcal{R}$ of values output by that failure detector. A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. As before, $H(p,t)$ is the value of the failure detector module of process p at time t. A *failure detector $\mathcal{D}$* is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ (where $\mathcal{R}_{\mathcal{D}}$ denotes the range of failure detector outputs of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

In Chapter 3, we considered a special class of failure detectors. Each failure detector module output a *set of processes* that are suspected to have crashed. In other words, for each failure detector $\mathcal{D}$, $\mathcal{R}_{\mathcal{D}} = 2^{\Pi}$.

We now formally define the failure detector $\diamond\mathcal{W}$ mentioned in the introduction. Each failure detector module of $\diamond\mathcal{W}$ outputs a set of processes that are suspected to have crashed: $\mathcal{R}_{\diamond\mathcal{W}} = 2^{\Pi}$. For each failure pattern F, $\diamond\mathcal{W}(F)$ is the set of all failure detector histories $H_{\diamond\mathcal{W}}$ with range $\mathcal{R}_{\diamond\mathcal{W}}$ that satisfy the following properties:

1. There is a time after which every process that crashes in F is always suspected by some process that is correct in F:

$$\exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \exists q \in \mathit{correct}(F), \forall t' \geq t : p \in H_{\diamond\mathcal{W}}(q, t')$$

2. There is a time after which some process that is correct in F is never suspected by any process that is correct in F:

$$\exists t \in \mathcal{T}, \exists p \in \mathit{correct}(F), \forall q \in \mathit{correct}(F), \forall t' \geq t : p \notin H_{\diamond\mathcal{W}}(q, t')$$

4.1.2 Algorithms

We model the asynchronous communication channels as a *message buffer* which contains messages of the form $(p, data, q)$ indicating that process p has sent *data* addressed to process q and q has not yet received that message. An *algorithm A* is a collection of n deterministic automata, one for each of the processes. $A(p)$ denotes the automaton running on process p. Computation proceeds in *steps of the given algorithm A*. In each step of A, process p performs atomically the following three phases:

Receive phase: p receives a single message of the form $(q, data, p)$ from the message buffer, or a "null" message, denoted λ, meaning that no message is received by p during this step.

Failure detector query phase: p queries and receives a value from its failure detector module. We say that p *sees a value* d when the value returned by p's failure detector module is d.

Send phase: p changes its state and sends a message to all the processes according to the automaton $A(p)$, based on its state at the beginning of the step, the message received in the receive phase, and the value that p sees in the failure detector query phase.[1]

The message actually received by the process p in the receive phase is chosen *non-deterministically* from amongst the messages in the message buffer destined to p, and the null message λ. The null message may be received *even if* there are messages in the message buffer that are destined to p: the fact that m is in the message buffer merely indicates that m was sent to p. Since ours will be a model of asynchronous systems, where messages may experience arbitrary (but finite) delays, the amount of time m may remain in the message buffer before it is received is unbounded. Indeed, our model will allow a message sent later than another to be received earlier than the other. Though message delays are arbitrary, we also want them to be finite. We model this by introducing a liveness assumption: every message sent will eventually be received, provided its recipient makes "sufficiently many" attempts to receive messages. All this will be made more precise later.

We also remark that the non-determinism arising from the choice of the message to be received reflects the asynchrony of the message buffer — it is not due to non-

[1] In the send phase, p sends a message to all the processes atomically. As was shown in [FLP85], the ability to do so is *not* sufficient for solving Consensus. An alternative formulation of a step could restrict a process to sending a message to a single process in the send phase. We can show that both formulations are equivalent for our purposes (see Section 4.6.1).

deterministic choices made by the process. The automaton $A(p)$ is deterministic in the sense that the message that p sends in a step and p's new state are uniquely determined from the present state of p, the message p received during the step and the failure detector value seen by p during the step.

To keep things simple we assume that a process p sends a message m to q at most once. This allows us to speak of the contents of the message buffer as a set, rather than a multiset. We can easily enforce this by adding a counter to each message sent by p to q — so this assumption does not damage generality.

4.1.3 Configurations, runs and environments

A *configuration* is a pair (s, M), where s is a function mapping each process p to its local state, and M is a set of triples of the form $(q, data, p)$ representing the messages presently in the message buffer. An *initial configuration of an algorithm* A is a configuration (s, M), where $s(p)$ is an initial state of $A(p)$ and $M = \emptyset$. A *step* of a given algorithm A transforms one configuration to another. A step of A is uniquely determined by the identity of the process p that takes the step, the message m received by p during that step, and the failure detector value d seen by p during the step. Thus, we identify a step of A with a tuple (p, m, d, A). If the message received in that step is the null message, then $m = \lambda$, otherwise m is of the type $(-, -, p)$. We say that a step $e = (p, m, d, A)$ *is applicable to a configuration* $C = (s, M)$ if and only if $m \in M \cup \{\lambda\}$. We write $e(C)$ to denote the unique configuration that results when e is applied to C.

A *schedule* S *of algorithm* A is a finite or infinite sequence of steps of A. $S_\perp$ denotes the empty schedule. We say that a schedule S of an algorithm A *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.[2] If S is a finite schedule applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C. Note

[2]We denote by $v[i]$ the ith element of a sequence v.

$S_\perp(C) = C$ for all configurations C. We say that C' *is a configuration of* (S, C) if there is a prefix S' of S such that $C' = S'(C)$.

A *partial run of algorithm* 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 A, S is a *finite* schedule of A, and T is a *finite* list of increasing time values (indicating when each step in S occurred) such that $|S| = |T|$, S is applicable to I, and for all $i \leq |S|$, if $S[i]$ is of the form (p, m, d, A) then:

- p has not crashed by time $T[i]$, i.e., $p \notin F(T[i])$
- d is the value of the failure detector module of p at time $T[i]$, i.e., $d = H_{\mathcal{D}}(p, T[i])$

Informally, a partial run of A using $\mathcal{D}$ represents a finite point of some execution of A using $\mathcal{D}$.

A *run of an algorithm* 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, $H_{\mathcal{D}} \in D(F)$ is a failure detector history, I is an initial configuration of A, S is an *infinite* schedule of A, and T is an *infinite* list of increasing time values indicating when each step in S occurred. In addition to satisfying the above properties of a partial run, a run must also satisfy the following properties:

- Every correct process takes an infinite number of steps in S. Formally:

$$\forall p \in \mathit{correct}(F), \forall i, \exists j > i : S[j] \text{ is of the type} (p, -, -, A)$$

- Every message sent to a correct process is eventually received. Formally:

$$\forall p \in \mathit{correct}(F), \forall C = (s, M) \text{ of } (S, I) : m = (q, \mathit{data}, p) \in M \Rightarrow$$
$$(\exists i : S[i] \text{ is of the type } (p, m, -, A))$$

In the Appendix, we prove that any algorithm that uses $\Diamond\mathcal{W}$ to solve Consensus requires $n > 2f$. With other failure detectors the requirements may be different. For example, there is a failure detector that can be used to solve Consensus only if p_1 and p_2 do not both crash. In general whether a given failure detector can be used to solve Consensus depends upon assumptions about the underlying "environment". Formally, an *environment* $\mathcal{E}$ (of an asynchronous system) is set of possible failure patterns.[3]

4.2 The Consensus problem

In this chapter, we study a weaker form of Consensus than that defined in Chapter 3. Since we prove a lower bound, this only strengthens our result. We first define this weaker Consensus and then describe how it differs from the previous version.

With the Consensus problem in this chapter, each process p has an *initial* value, 0 or 1, and must reach an irrevocable decision on one of these values. Thus, the algorithm of process p, $A(p)$, has two distinct *initial states* σ_0^p and σ_1^p signifying that p's initial value is 0 or 1. $A(p)$ also has two disjoint sets of *decision states* Σ_0^p and Σ_1^p. If p enters a state in Σ_k^p, we say that *p has decided k*.

We say that algorithm A *uses failure detector $\mathcal{D}$ to solve Consensus in environment $\mathcal{E}$* if every run $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ of A using $\mathcal{D}$ where $F \in \mathcal{E}$ satisfies:

Termination: Every correct process eventually decides some value. Formally:

$$\forall p \in \mathit{correct}(F), \exists C = (s, M) \text{ of } (S, I) : s(p) \in \Sigma_0^p \cup \Sigma_1^p$$

Validity: If a correct process decides v, then v was proposed by some process. Formally, let $I = (s_0, M_0)$:

[3] In a synchronous system, assumptions about the underlying environment may also include other characteristics such as the relative process speeds, the maximum message delay, the degree of clock synchronization, etc. In such a system, a more elaborate definition of an environment would be required.

$$\forall p \in \mathit{correct}(F), \forall k \in \{0,1\} : (\exists C = (s, M) \text{ of } (S, I) : \\ s(p) \in \Sigma_k^p) \Rightarrow (\exists q \in \Pi : s_0(q) = \sigma_k^q)$$

Uniform Integrity: Every process decides at most once. Formally:

$$\forall p \in \Pi, \forall k \in \{0,1\}, \forall i, S[i](I) = (s, M), \forall i' > i, S[i'](I) = (s', M') : \\ s(p) \in \Sigma_k^p \Rightarrow s'(p) \in \Sigma_k^p$$

Agreement: No two correct processes decide differently. Formally:

$$\forall p, p' \in \mathit{correct}(F), \forall C = (s, M) \text{ of } (S, I), \forall k, k' \in \{0,1\} : \\ (s(p) \in \Sigma_k^p \wedge s(p') \in \Sigma_{k'}^{p'}) \Rightarrow k = k'$$

The Consensus problem defined above differs from the previous chapter's version in the following ways:

- In the previous chapter, Consensus is *multi-valued*, i.e., each process is allowed to propose a value from an arbitrary range. In this chapter, Consensus is *binary*, i.e., each process can only propose 0 or 1. It is not difficult to show that both versions of Consensus are equivalent (i.e., reducible to each other) in asynchronous systems.
- The *validity* property of the above Consensus imposes a requirement on the decision value of *correct* processes only. In contrast, the *uniform validity* property of the previous version of Consensus also imposed this requirement on every faulty process that decides.
- In the previous chapter, processes could execute several independent instances of Consensus. For each such instance, every process makes a distinct proposal and reaches a corresponding decision.[4] In this chapter, processes

[4]The ability to solve "repeated" Consensus was crucial to the reduction of Atomic Broadcast to Consensus (see the algorithm in Figure 3.7).

execute a *single* instance of Consensus. Thus, we can model the *unique* proposal of each process as its initial value.

4.3 Reducibility

We now define what it means for an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ to transform a failure detector $\mathcal{D}$ into another failure detector $\mathcal{D}'$ in an environment $\mathcal{E}$. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ to maintain a variable $output_p$ at every process p. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ *in* $\mathcal{E}$ if and only if for every run $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, where $F \in \mathcal{E}$, $output^R \in \mathcal{D}'(F)$. If there is an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ into $\mathcal{D}'$ in $\mathcal{E}$, we write $\mathcal{D} \succeq_{\mathcal{E}} \mathcal{D}'$ and say that $\mathcal{D}'$ *is reducible to* $\mathcal{D}$ *in* $\mathcal{E}$; we also say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$.

4.4 An outline of the result

In Section 3.5.2 we showed that $\diamond\mathcal{W}$ can be used to solve Consensus in any environment in which $n > 2f$. We now show that $\diamond\mathcal{W}$ is weaker than any failure detector that can be used to solve Consensus. This result holds for any environment $\mathcal{E}$. Together with the result in Section 3.5.2, this implies that $\diamond\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in any environment in which $n > 2f$.

To prove our result, we first define a new failure detector, denoted Ω, that is at least as strong as $\diamond\mathcal{W}$. We then show that any failure detector $\mathcal{D}$ that can be used to solve Consensus is at least as strong as Ω. Thus, $\mathcal{D}$ is at least as strong as $\diamond\mathcal{W}$.

The output of the failure detector module of Ω at a process p is a *single* process, q, that p currently considers to be *correct*; we say that p *trusts* q. In this case, $\mathcal{R}_\Omega = \Pi$. For each failure pattern F, $\Omega(F)$ is the set of all failure detector histories H_Ω with range $\mathcal{R}_\Omega$ that satisfy the following property:

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

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

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

Theorem 35: For all environments $\mathcal{E}$, $\Omega \succeq_{\mathcal{E}} \diamond\mathcal{W}$.

PROOF: [Sketch] The reduction algorithm $T_{\Omega \rightarrow \diamond\mathcal{W}}$ that transforms Ω into $\diamond\mathcal{W}$ is as follows. Each process p periodically sets $\mathit{output}_p \leftarrow \Pi - \{q\}$, where q is the process that p currently trusts according to Ω. It is easy to see that (in any environment $\mathcal{E}$) this output satisfies the two properties of $\diamond\mathcal{W}$. $\square$

Theorem 36: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \Omega$.

PROOF: The reduction algorithm $T_{\mathcal{D} \rightarrow \Omega}$ is shown in Section 4.5. It is the core of our result. $\square$

Corollary 37: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \diamond\mathcal{W}$.

PROOF: If $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then, by Theorem 36, $\mathcal{D} \succeq_{\mathcal{E}} \Omega$. From Theorem 35, $\Omega \succeq_{\mathcal{E}} \diamond\mathcal{W}$. By transitivity, $\mathcal{D} \succeq_{\mathcal{E}} \diamond\mathcal{W}$. $\square$

In Section 3.5.2 we proved that, for all environments $\mathcal{E}$ in which $n > 2f$, $\diamond\mathcal{W}$ can be used to solve Consensus. Together with Corollary 37, this shows that:

Theorem 38: For all environments $\mathcal{E}$ in which $n > 2f$, $\diamond\mathcal{W}$ is the weakest failure detector that can be used to solve Consensus in $\mathcal{E}$.

4.5 The reduction algorithm

Let $\mathcal{E}$ be an environment, $\mathcal{D}$ be a failure detector that can be used to solve Consensus in $\mathcal{E}$, and $\mathit{Consensus}_{\mathcal{D}}$ be the Consensus algorithm that uses $\mathcal{D}$. We describe an algorithm $T_{\mathcal{D}\to\Omega}$ that transforms $\mathcal{D}$ into Ω in $\mathcal{E}$. Intuitively, this algorithm works as follows. Fix an arbitrary run of $T_{\mathcal{D}\to\Omega}$ using $\mathcal{D}$ in $\mathcal{E}$, with failure pattern $F \in \mathcal{E}$, and failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$. We shall first construct an infinite directed acyclic graph, denoted G, whose vertices are some of the failure detector values that occur in $H_{\mathcal{D}}$, and whose edges are consistent with the time at which these values occur. We then show that G induces a simulation forest Υ that encodes an infinite set of possible runs of $\mathit{Consensus}_{\mathcal{D}}$. Finally, we show how to extract from Υ the identity of a process p^* that is correct in F.

The induced simulation forest is infinite and thus cannot be computed by any process. However, the information needed to extract p^* is present in a *finite* subgraph of the forest. It will be sufficient for each correct process p to construct ever increasing finite approximations of the simulation forest Υ that will eventually include this crucial finite subgraph. At all times, p uses its present approximation of Υ to select the identity of some process: once p's approximation of Υ includes the crucial finite subgraph, the selected process will be p^* (forever). Thus, there is a time after which all correct processes trust the same correct process, p^*—which is exactly what Ω requires.

We say that a process is *correct* (*crashes*) if it is correct (crashes) in F. For simplicity, we assume that a process p sees a value d at most once (this can be enforced by tagging a counter to each value seen). For the rest of this chapter, whenever we refer to a run of $\mathit{Consensus}_{\mathcal{D}}$, we mean a run of $\mathit{Consensus}_{\mathcal{D}}$ using $\mathcal{D}$. Furthermore, we only consider schedules of $\mathit{Consensus}_{\mathcal{D}}$, and therefore we write (p, m, d) instead of $(p, m, d, \mathit{Consensus}_{\mathcal{D}})$ to denote a step.

4.5.1 A DAG and a forest

Given the failure pattern F and the corresponding failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$ that were fixed above, let G be any infinite directed acyclic graph with the following properties:

1. The vertices of G are of the form $[p, d]$ where $p \in \Pi$ and $d \in \mathcal{R}_{\mathcal{D}}$. If $[p, d]$ is a vertex of G, then there is a time t such that $p \notin F(t)$ and $d = H_{\mathcal{D}}(p, t)$ (i.e., at time t, p has not crashed and the value of p's failure detector module is d).

2. If $[q_1, d_1] \rightarrow [q_2, d_2]$ is an edge of G and $d_1 = H_{\mathcal{D}}(q_1, t_1)$ and $d_2 = H_{\mathcal{D}}(q_2, t_2)$ then $t_1 < t_2$.

3. G is transitively closed.

4. Let p be any correct process and V be a finite subset of vertices of G. There is a failure detector value d such that for all vertices $[p', d']$ in V, $[p', d'] \rightarrow [p, d]$ is an edge of G.

Note that such a DAG represents only a "sampling" of the failure detector values that occur in $H_{\mathcal{D}}$. In particular, we do not require that it contain all the values that occur in $H_{\mathcal{D}}$ or that it relate (with an edge) all the values according to the time at which they occur. However, Property 4 implies that the DAG contains infinitely many "samplings" of the failure detector module of each *correct* process.

Lemma 39: Let V be any finite subset of vertices in G. G has an infinite path g such that:

- There is an edge from every vertex of V to the first vertex of g.

- If $[p, -]$ is a vertex of g then p is correct; for each correct p, there are infinitely many vertices $[p, -]$ in g.

PROOF: By repeated application of Property 4. □

Let $g = [q_1, d_1], [q_2, d_2], \ldots$ be any (finite or infinite) path of G. A schedule S is *compatible with* g if it has the same length as g, and $S = (q_1, m_1, d_1), (q_2, m_2, d_2), \ldots$, for some (possibly null) messages $m_1, m_2, \ldots$ We say that S is *compatible with* G if it is compatible with some path of G.

Let I be any initial configuration of $Consensus_{\mathcal{D}}$. We define the *simulation tree* Υ_G^I *induced by* G *and* I as follows. The vertices of Υ_G^I are the *finite* schedules S that are compatible with G and are applicable to I. The root of Υ_G^I is the empty schedule $S_\perp$. There is an edge from vertex S to vertex S' if and only if $S' = S \cdot e$ for a step e;[5] this edge is labeled e. With each (finite or infinite) path in Υ_G^I, we associate the unique schedule $S = e_1, e_2, \ldots, e_k, \ldots$ consisting of the sequence of labels of the edges on that path. Note that if a path starts from the root of Υ_G^I and it is finite, the schedule S associated with it is also the last vertex of that path.

Lemma 40: S is a schedule associated with a path of Υ_G^I that starts from the root if and only if S is a schedule compatible with G and applicable to I.

PROOF: The lemma obviously holds if S is a *finite* schedule (this is immediate from the definitions). Now let $S = e_1, e_2, \ldots, e_i, \ldots$ be an *infinite* schedule, where $e_i = [q_i, m_i, d_i]$. We define $S_0 = S_\perp$, $S_1 = e_1$, $S_2 = S_1 \cdot e_2$, and in general $S_i = S_{i-i} \cdot e_i$ for all $i = 1, 2, \ldots$

Assume that S is compatible with G and applicable to I. We must show that S is a schedule associated with a path of Υ_G^I that starts from the root. To see this, note that for all $i \geq 0$, S_i is a finite schedule that is also compatible with G and applicable to I. Thus, all the schedules $S_0, S_1, S_2, \ldots, S_{i-1}, S_i, \ldots$ are vertices of Υ_G^I. Since $S_i = S_{i-1} \cdot e_i$, the edge from S_{i-1} to S_i is labeled e_i, for all $i \geq 1$. Thus, $S = e_1, e_2, \ldots, e_i, \ldots$ is the schedule associated with the infinite path $S_0 \to S_1 \to S_2 \to \ldots \to S_{i-1} \to S_i \to \ldots$ of Υ_G^I; this path starts from the root

[5] If u, w are sequences and v is finite then $v \cdot w$ denotes the concatenation of the two sequences.

$S_0 = S_\perp$.

Assume that S is a schedule associated with an infinite path of Υ_G^I that starts from the root. We must show that S is compatible with G and is applicable to I. First note that for all i, S_i is a vertex in Υ_G^I, thus S_i is compatible with G and is applicable to I. Since $S_i = [q_1, m_1, d_1], [q_2, m_2, d_2], \ldots, [q_i, m_i, d_i]$ is compatible with G, G must contain the path $\pi_i = [q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i]$ (for all i). Note that, for all i, $\pi_{i+1} = \pi_i \cdot [q_{i+1}, d_{i+1}]$ is an extension of the path π_i in G. Therefore, G contains the *infinite* path $[q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i], \ldots$ So S is compatible with G. Furthermore, since all S_i's are applicable to I, by definition of applicability, the infinite schedule S is also applicable to I. Thus, S is compatible with G and applicable to I. □

The following two lemmata show that the finite and infinite paths of Υ_G^I correspond to partial runs and runs of *Consensus*$_\mathcal{D}$ with initial configuration I.

Lemma 41: Let S be a schedule associated with a *finite* path of Υ_G^I that starts from the root. There is a sequence of times T such that $\langle F, H_\mathcal{D}, I, S, T\rangle$ is a partial run of *Consensus*$_\mathcal{D}$.

Proof: By Lemma 40, S is applicable to I and compatible with G. Thus S is compatible with some finite path $g = [q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i], \ldots, [q_k, d_k]$ of G. From Property 1 of G (applied to every vertex of the path g), there is a sequence $T = t_1, t_2, \ldots, t_i, \ldots, t_k$ of times such that for all i, $1 \le i \le k$, $d_i = H_\mathcal{D}(q_i, t_i)$ and $q_i \notin F(t_i)$. From Property 2 of G (applied to every edge of the path g), for all i, $1 \le i < k$, $t_i < t_{i+1}$. Thus T is a sequence of increasing times, and, by definition, $\langle F, H_\mathcal{D}, I, S, T\rangle$ is a partial run of *Consensus*$_\mathcal{D}$. □

Lemma 42: Let S be a schedule associated with an *infinite* path of Υ_G^I that starts from the root. If in S every correct process takes an infinite number of steps and every message sent to a correct process is eventually received, there is a sequence of times T such that $\langle F, H_\mathcal{D}, I, S, T\rangle$ is a run of *Consensus*$_\mathcal{D}$.

PROOF: Similar to Lemma 41. □

The following lemmata show some "richness" properties of the simulation trees induced by G.

Lemma 43: For any two initial configurations I and I', if S is a vertex of Υ_G^I and is applicable to I' then S is also a vertex of $\Upsilon_G^{I'}$.

PROOF: Follows directly from the definitions. □

Lemma 44: Let S be any vertex of Υ_G^I and p be any correct process. Let m be a message in the message buffer of $S(I)$ addressed to p or the null message. For some d, S has a child $S \cdot (p, m, d)$ in Υ_G^I.

PROOF: From the definition of Υ_G^I, S is compatible with some finite path g of G and applicable to I. Let v denote the last vertex of g. By Property 4, there is a d such that $v \rightarrow [p, d]$ is an edge of G. Therefore, $g \cdot [p, d]$ is a path of G, and $S \cdot (p, m, d)$ is compatible with G.

It remains to show that $S \cdot (p, m, d)$ is applicable to I. Since S is applicable to I, it suffices to show that (p, m, d) is applicable to $S(I)$. But this is true since, by hypothesis, m is in the message buffer of $S(I)$ and addressed to p, or the null message. □

Lemma 45: Let S be any vertex of Υ_G^I and p be any process. Let m be a message in the message buffer of $S(I)$ addressed to p or the null message. Let S' be a descendent of S such that, for some d, $S' \cdot (p, m, d)$ is in Υ_G^I. For each vertex S'' on the path from S to S' (inclusive), $S'' \cdot (p, m, d)$ is also in Υ_G^I.

PROOF: Since they are vertices of Υ_G^I, S, S'' and $S' \cdot (p, m, d)$ are compatible with some finite paths g, $g \cdot g''$ and $g \cdot g'' \cdot g' \cdot [p, d]$ of G, respectively. From Property 3 (transitive closure) of G, $g \cdot g'' \cdot [p, d]$ is also a path of G. So $S'' \cdot (p, m, d)$ is compatible with this path of G. We now show that $S'' \cdot (p, m, d)$ is also applicable to I, and therefore it is a vertex of Υ_G^I.

Since S'' is a vertex of Υ_G^I, S'' is applicable to I. If $m = \lambda$, then (p, m, d) is obviously applicable to $S''(I)$. Now suppose $m \neq \lambda$. Since $S' \cdot (p, m, d)$ is a vertex of Υ_G^I, (p, m, d) is applicable to $S'(I)$, and thus m is in the message buffer of $S'(I)$. Since each message is sent at most once and m is in the message buffers of $S(I)$ and $S'(I)$, there is no edge of the type $(p, m, -)$ on the path from S to S'. So m is also in the message buffer of $S''(I)$, and (p, m, d) is applicable to $S''(I)$. □

Lemma 46: Let S, S_0, and S_1 be any vertices of Υ_G^I. There is a finite schedule E containing only steps of correct processes such that:

1. $S \cdot E$ is a vertex of Υ_G^I and all correct processes have decided in $S \cdot E(I)$.
2. For $i = 0, 1$, if E is applicable to $S_i(I)$ then $S_i \cdot E$ is a vertex of Υ_G^I.

PROOF: Since S is a vertex of Υ_G^I, S is compatible with some finite path g of G and is applicable to I. Similarly, S_0 and S_1 are compatible with some finite path g_0 and g_1, respectively, of G. From Lemma 39 (applied to the last vertices of g, g_0 and g_1), G has an infinite path $g_\infty = [q_1, d_1], [q_2, d_2], \ldots, [q_j, d_j], \ldots$ with the following two properties:

1. There is an edge from the last vertex of g, g_0 and g_1 to the first vertex of g_∞. (Thus, $g \cdot g_\infty$, $g_0 \cdot g_\infty$, and $g_1 \cdot g_\infty$ are infinite paths in G.)
2. If $[p, -]$ is a vertex of g_∞ then p is correct; for each correct p, there are infinitely many vertices $[p, -]$ in g_∞.

We now show how to construct the required schedule E. Consider the infinite sequence of schedules $S^0, S^1, S^2, \ldots, S^j, \ldots$ constructed by the algorithm in Figure 4.1. An easy induction shows that for all $j > 0$, S^j is applicable to I and is compatible with $g \cdot [q_1, d_1] \cdot \ldots \cdot [q_j, d_j]$, a prefix of the path $g \cdot g_\infty$ in G. So, for all $j > 0$, S^j is a vertex of Υ_G^I. Consider the infinite path of Υ_G^I that starts from the root of Υ_G^I then goes to $S^0 = S$, and then to $S^1, S^2, \ldots, S^j, \ldots$ The infinite

$j \leftarrow 0$
$S^0 \leftarrow S$ $\{S^0$ *is compatible with* g *and applicable to* $I\}$
repeat forever
$j \leftarrow j+1$
Let $[q_j, d_j]$ be the j-th vertex of path g_∞
Let m_j be the oldest message addressed to q_j in the message buffer of $S^{j-1}(I)$
(if no such message exists, $m_j = \lambda$)
$e_j \leftarrow (q_j, m_j, d_j)$
$S^j \leftarrow S^{j-\mathbf{1}} \cdot e_j$ $\{S^j$ *is compatible with* $g \cdot [q_1, d_1] \cdot \ldots \cdot, [q_j, d_j]$ *and applicable to* $I\}$

Figure 4.1: Generating schedule $S \cdot E^\infty$, compatible with path $g \cdot g_\infty$, in Υ_G^I

schedule associated with that path is $S^\infty = S \cdot e_1 \cdot e_2 \cdot \ldots \cdot e_j \ldots$ Note that schedule $E^\infty = e_1 \cdot e_2 \cdot \ldots \cdot e_j \ldots$ is compatible with path g_∞ of G. By Property (2) of path g_∞, every correct process p takes an infinite number of steps in E^∞ (and thus also in $S^\infty = S \cdot E^\infty$). Since in each one of these steps p receives the oldest message that is addressed to it, every message sent to p (in S^∞) is eventually received. By Lemma 42, there is a T such that $R = \langle F, H_{\mathcal{D}}, I, S^\infty, T\rangle$ is a run of *Consensus*$_{\mathcal{D}}$.

From the termination requirement of Consensus, S^∞ has a finite prefix S^d such that all correct processes have decided in $S^d(I)$. There are two cases:

- S^d is a prefix of S. Since decisions are irrevocable, all correct processes remain decided in $S(I)$. Thus $S_\perp$, the empty schedule, is the required E.

- S is a prefix of S^d. Thus, $S^d = S \cdot E$ where E is a finite prefix of E^∞. Since E^∞ is compatible with g_∞, E is compatible with a prefix of g_∞. Now consider S_0 (the following argument also applies to S_1). Since S_0 is compatible with g_0, $S_0 \cdot E$ is compatible with a prefix of $g_0 \cdot g_\infty$, a path in G. So, $S_0 \cdot E$ is compatible with G. If $S_0 \cdot E$ is also applicable to I, then, by the definition of Υ_G^I, it is a vertex of Υ_G^I. The same argument holds for S_1. It remains to show that E contains only steps of correct processes. This is immediate from Property (2) of g_∞ and from the fact that E is compatible with a prefix of g_∞. $\square$

Let I^i, $0 \leq i \leq n$, denote the initial configuration of $Consensus_{\mathcal{D}}$ in which the initial values of $p_1 \ldots p_i$ are 1, and the initial values of $p_{i+1} \ldots p_n$ are 0. The *simulation forest induced by G* is the set $\{\Upsilon_G^{I^0}, \Upsilon_G^{I^1}, \ldots, \Upsilon_G^{I^n}\}$ of simulation trees induced by G and initial configurations $I^0, I^1, \ldots, I^n$.

4.5.2 Tagging the simulation forest

We assign a set of *tags* to each vertex of every tree in the simulation forest induced by G. Vertex S of tree Υ_G^I gets tag k if and only if it has a descendent S' such that some *correct* process has decided k in $S'(I)$. Hereafter, Υ^i denotes the tagged tree $\Upsilon_G^{I^i}$, and Υ denotes the tagged simulation forest $\{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$.

Lemma 47: Every vertex of Υ^i has at least one tag.

PROOF: From Lemma 46, every vertex S of Υ^i has a descendent $S' = S \cdot E$ (for some E) such that all correct processes have decided in $S'(I^i)$. $\square$

A vertex of Υ^i is *monovalent* if it has only one tag, and *bivalent* if it has both tags, 0 and 1. A vertex is *0-valent* if it is monovalent and is tagged 0; *1-valent* is similarly defined.

Lemma 48: Every vertex of Υ^i is either 0-valent, 1-valent, or bivalent.

PROOF: Immediate from Lemma 47. $\square$

Lemma 49: The ancestors of a bivalent vertex are bivalent. The descendents of a k-valent vertex are k-valent.

PROOF: Immediate from the definitions. $\square$

Lemma 50: If vertex S of Υ^i has tag k, then no correct process has decided $1-k$ in $S(I^i)$.

PROOF: Since S has tag k, it has a descendent S' such that a correct process p has decided k in $S'(I^i)$. From Lemma 41, there is a T such that $R = \langle F, H_{\mathcal{D}}, I^i, S', T\rangle$

is a partial run of $Consensus_{\mathcal{D}}$. Since p has decided k in $S'(I^i)$, from the agreement requirement of Consensus, no correct process has decided $1-k$ in $S'(I^i)$. Since S' is a descendent of S, no correct process could have decided $1-k$ in $S(I^i)$. □

Lemma 51: If vertex S of Υ^i is bivalent, then no correct process has decided in $S(I^i)$.

PROOF: Immediate from Lemma 50. □

Recall that in I^0 all processes have initial value 0, while in I^n they all have initial value 1.

Lemma 52: The root of Υ^0 is 0-valent; the root of Υ^n is 1-valent.

PROOF: We first show that the root of Υ^0 is 0-valent. Suppose, for contradiction, that the root of Υ^0 has tag 1. There must be a vertex S of Υ^0 such that some correct process has decided 1 in $S(I^0)$. From Lemma 41, there is a T such that $R = \langle F, H_{\mathcal{D}}, I^0, S, T\rangle$ is a partial run of $Consensus_{\mathcal{D}}$. R violates the validity requirement of Consensus—a contradiction. Thus the root of Υ^0 cannot have a tag of 1. From Lemma 47, the root of Υ^0 has at least one tag: thus it is 0-valent.

By a symmetric argument, the root of Υ^n is 1-valent. □

Index i is *critical* if the root of Υ^i is bivalent, or if the root of Υ^{i-1} is 0-valent while the root of Υ^i is 1-valent. In the first case, we say that index i is *bivalent critical*; in the second case, we say that i is *monovalent critical.*

Lemma 53: There is a critical index i, $0 < i \leq n$.

PROOF: Apply Lemmata 48 and 52 to the roots of Υ^0, $\Upsilon^1, \ldots, \Upsilon^n$. □

The critical index i is the key to extracting the identity of a correct process. In fact, if i is monovalent critical, we shall prove that p_i must be correct (Lemma 55).

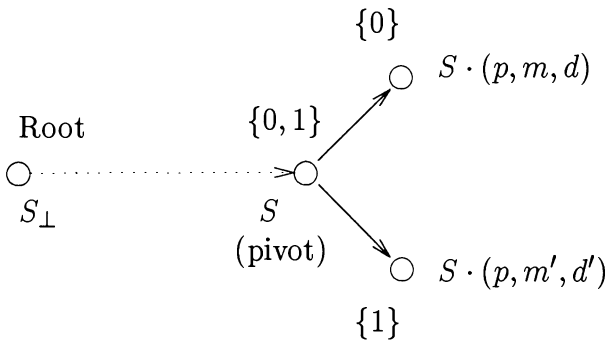

Figure 4.2: A fork—p is the deciding process

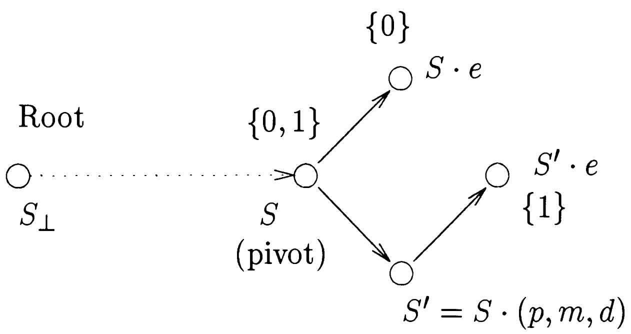

Figure 4.3: A hook—p is the deciding process

If i is bivalent critical, the correct process will be found by focusing on the tree Υ^i, as explained in the following section.

4.5.3 Of hooks and forks

We describe two types of finite subtrees of Υ^i referred to as *decision gadgets of* Υ^i. Each type of decision gadget is rooted at the root $S_\perp$ of Υ^i and has exactly two leaves: one 0-valent and one 1-valent. The least common ancestor of these leaves is called the *pivot*. The pivot is clearly bivalent.

The first type of decision gadget is called a *fork*, and is shown in Figure 4.2.

The two leaves are children of the pivot, obtained by applying different steps of the *same* process p. Process p is the *deciding process of the fork*, because its step after the pivot determines the decision of correct processes.

The second type of decision gadget is called a *hook*, and is shown in Figure 4.3. Let S be the pivot of the hook. There is a step e such that $S \cdot e$ is one leaf, and the other leaf is $S \cdot (p, m, d) \cdot e$ for some p, m, d. Process p is the *deciding process of the hook*, because the decision of correct processes is determined by whether p takes the step (p, m, d) before e.

We shall prove that the deciding process p of a decision gadget must be correct (Lemma 57). Intuitively, this is because if p crashes no process can figure out whether p has taken the step that determines the decision value. The existence of such a critical "hidden" step is also at the core of many impossibility proofs starting with [FLP85]. In our case, the "hiding" is more difficult because now processes have recourse to the failure detector $\mathcal{D}$. Despite this, the hiding of the step of the deciding process of a decision gadget is still possible.

Lemma 54: If index i is bivalent critical then Υ^i has at least one decision gadget (and hence a deciding process).

PROOF: Starting from the bivalent root of Υ^i, we generate a path π in Υ^i, all the vertices of which are bivalent, as follows. We consider all correct processes in round-robin fashion. Suppose we have generated path S so far, and it is the turn of process p. Let m be the the oldest message destined to p that is in the message buffer of $S(I^i)$.[6] (If no such message exists, we take m to be the null message.) We try to extend the path S so that the last edge in the extension corresponds to p receiving m and the target of that edge is a bivalent vertex. The path construction ends if and when such an extension is no longer possible. This construction is shown in Figure 4.4. Each iteration of the loop extends the path

[6] By a slight abuse of notation we identify a finite path from the root and its associated schedule.

$S \leftarrow S_\perp$ $\{S_\perp$ *is the bivalent root of* $\Upsilon^i\}$
repeat forever
  Let p be the next correct process in round-robin order
  Let m be the oldest message addressed to p in the message buffer of $S(I^i)$
    (if no such message exists, $m = \lambda$)
  if S has a descendent S' such that, for some d, $S' \cdot (p, m, d)$ is a bivalent
    vertex in Υ^i **then** $S \leftarrow S' \cdot (p, m, d)$ $\{S$ *is bivalent*$\}$
  else exit

Figure 4.4: Generating path π in Υ^i

by at least one edge. Let π be the path generated by these iterations; π is finite or infinite depending on whether the loop terminates.

Claim 1: π is finite.

PROOF: Suppose, for contradiction, that π is infinite. Let S be the schedule associated with π. By construction, in S every correct process takes an infinite number of steps and every message sent to a correct process is eventually received. By Lemma 42, there is a T such that $R = \langle F, H_\mathcal{D}, I^i, S, T\rangle$ is a run of *Consensus*$_\mathcal{D}$. By construction, all vertices in π are bivalent. By Lemma 51, no correct process decides in R, thus violating the termination requirement of Consensus—a contradiction. $\Box_{\textbf{claim 1}}$

Let S be the last vertex of π (clearly, S is bivalent). Let p be the next correct process in round-robin order when the loop in Figure 4.4 terminates. Let m be the oldest message addressed to p in the message buffer of $S(I^i)$ (if no such message exists, m is the null message). The loop exit condition and Lemma 48 imply that

$$\text{All descendents } S' \cdot (p, m, -) \text{ of } S \text{ are monovalent.} \qquad (*)$$

From Lemma 44, for some d, S has a child $S \cdot (p, m, d)$ in Υ^i. By $(*)$, $S \cdot (p, m, d)$ is monovalent. Without loss of generality, assume it is 0-valent.

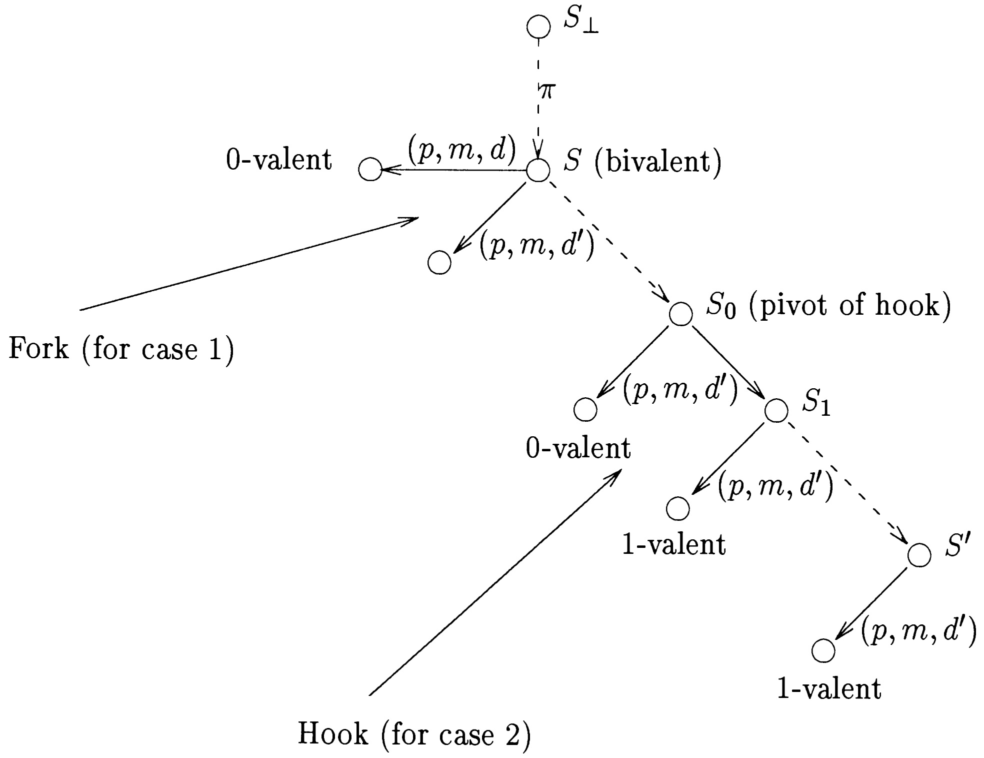

Figure 4.5: The decision gadgets in Υ^i if i is bivalent critical

Claim 2: For some d' there is a descendent S' of S such that $S' \cdot (p, m, d')$ is a 1-valent vertex of Υ^i, and the path from S to S' contains no edge labeled $(p, m, -)$.

PROOF: Since S is bivalent, it has a descendent S^* such that some correct process has decided 1 in $S^*(I^i)$. From Lemmata 47 and 50, S^* is 1-valent. There are two cases:

1. The path from S to S^* does not have an edge labeled $(p, m, -)$. Suppose $m \neq \lambda$. Since m is in the message buffer of $S(I^i)$ and p does not receive m in the path from S to S^*, m is still in the message buffer of $S^*(I^i)$. From Lemma 44, for some d', $S^* \cdot (p, m, d')$ is in Υ^i. Since S^* is 1-valent, by Lemma 49, $S^* \cdot (p, m, d')$ is also 1-valent. In this case, the required S' is S^*.

2. The path from S to S^* has an edge labeled $(p, m, -)$. Let (p, m, d') be the first such edge on that path. Let S' be the source of this edge. By $(*)$, $S' \cdot (p, m, d')$ is monovalent. Since $S' \cdot (p, m, d')$ has a 1-valent descendent S^*, by Lemma 49, $S' \cdot (p, m, d')$ is 1-valent. $\square_{\textbf{claim 2}}$

Consider the vertex S' and edge (p, m, d') of Claim 2. By Lemma 45, for each vertex S'' on the path from S to S' (inclusive), $S'' \cdot (p, m, d')$ is also in Υ^i. By $(*)$, all such vertices $S'' \cdot (p, m, d')$ are monovalent. In particular, $S \cdot (p, m, d')$ is monovalent. There are two cases (see Figure 4.5):

1. $S \cdot (p, m, d')$ is 1-valent. Since $S \cdot (p, m, d)$ is 0-valent, Υ^i has a fork with pivot S.

2. $S \cdot (p, m, d')$ is 0-valent. Recall that $S' \cdot (p, m, d')$ is 1-valent and for each vertex S'' between S and S', $S'' \cdot (p, m, d')$ is monovalent. Thus, the path from S to S' must have two vertices S_0 and S_1 such that S_0 is the parent of S_1, $S_0 \cdot (p, m, d')$ is 0-valent and $S_1 \cdot (p, m, d')$ is 1-valent. Hence, Υ^i has a hook with pivot S_0. $\square$

4.5.4 Extracting the correct process

By Lemma 53, there is a critical index i. If i is monovalent critical, Lemma 55 below shows how to extract a correct process. If i is bivalent critical, a correct process can be found by applying Lemmata 54 and 57.

Lemma 55: If index i is monovalent critical then p_i is correct.

PROOF: Suppose, for contradiction, that p_i crashes. By Lemma 46(1) (applied to the root $S = S_\perp$ of Υ^i), there is a finite schedule E that contains only steps of correct processes (and hence no step of p_i) such that all correct processes have decided in $E(I^i)$. Since index i is monovalent critical, the root $S_\perp$ of Υ^i is 1-valent. Hence all correct processes must have decided 1 in $E(I^i)$.

I^i and I^{i-1} only differ in the state of p_i. Since S is applicable to I^i and does not contain any steps of p_i, an easy induction on the number of steps in S shows that: (a) S is also applicable to I^{i-1}, and (b) the state of all processes other than p_i are the same in $S(I^i)$ and $S(I^{i-1})$. Using Lemma 43, (a) implies that S is also a vertex of Υ^{i-1}. By (b), all correct processes have decided 1 in $S(I^{i-1})$. Thus the root of Υ^{i-1} has tag 1. Since i is monovalent critical, the root of Υ^{i-1} is 0-valent—a contradiction. □

Lemma 56: Let S be any bivalent vertex of Υ^i, and S_0, S_1 be any 0-valent and 1-valent descendents of S. If the paths from S to S_0 and from S to S_1 contain only steps of the form $(p, -, -)$, then p is correct.

PROOF: Suppose, for contradiction, that p crashes. From Lemma 46, there is a schedule E containing only steps of correct processes (and hence no step of p) such that:

1. $S \cdot E$ is a vertex of Υ^i and all correct processes have decided in $S \cdot E(I^i)$.

2. For $k = 0, 1$, if $S_k \cdot E$ is applicable to I^i then $S_k \cdot E$ is a vertex of Υ^i.

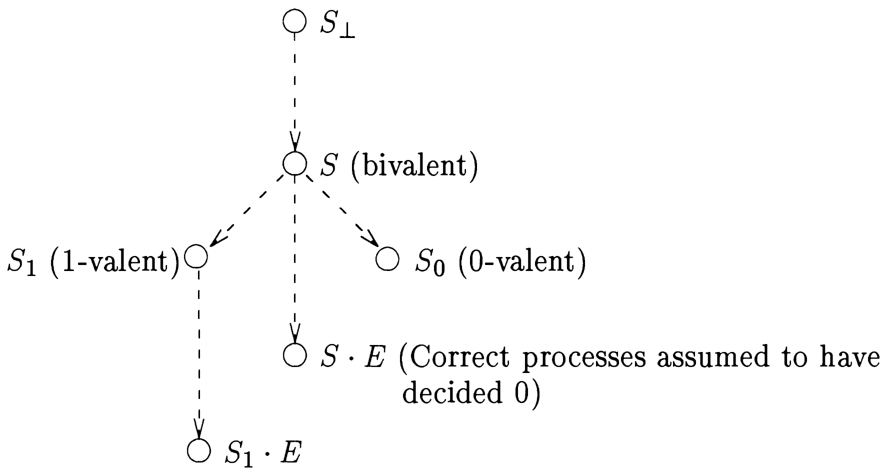

Figure 4.6: Lemma 53

Without loss of generality assume that all correct processes decided 0 in $S \cdot E(I^i)$. Refer to Figure 4.6. Since all steps in the path from S to S_1 are steps of p, the state of every process other than p is the same in $S(I^i)$ and in $S_1(I^i)$. Furthermore, any message addressed to a process other than p that is in the message buffer in $S(I^i)$ is still in the message buffer in $S_1(I^i)$. Since E is applicable to $S(I^i)$ and does not contain any steps of p, an easy induction on the number of steps in E shows that: (a) E is also applicable to $S_1(I^i)$, and (b) the state of every process other than p is the same in $S \cdot E(I^i)$ and $S_1 \cdot E(I^i)$. By (ii), (a) implies that $S_1 \cdot E(I^i)$ is a vertex in Υ^i. By (b), all correct processes decide 0 in $S_1 \cdot E(I^i)$. Thus S_1, has tag 0. But S_1 is 1-valent—a contradiction. $\square$

Lemma 57: The deciding process of a decision gadget is correct.

PROOF: Let γ be any decision gadget of Υ^i. There are two cases to consider:

1. γ is a fork. By Lemma 56, the deciding process of γ is correct.

2. γ is a hook. Assume (without loss of generality) that S is the pivot of γ, $S_0 = S \cdot (p', m', d')$ is the 0-valent leaf of γ and $S_1 = S \cdot (p, m, d) \cdot (p', m', d')$

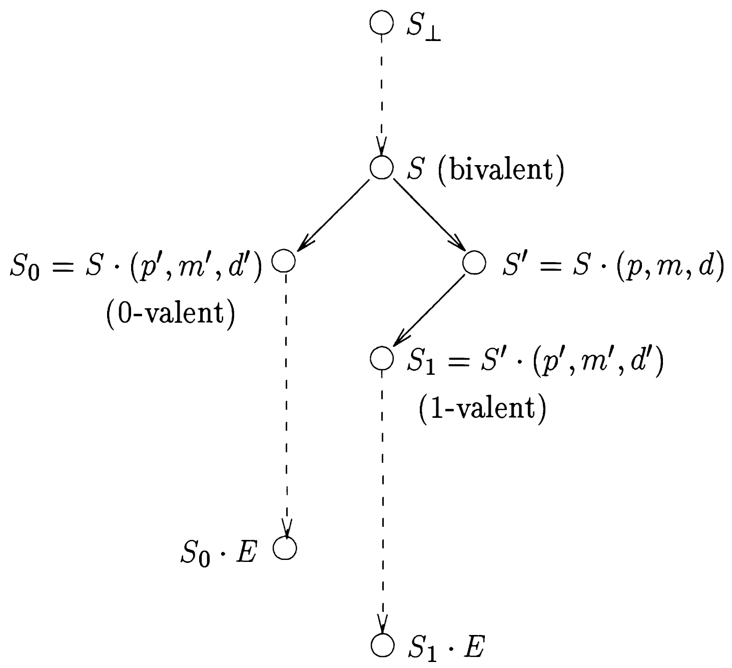

Figure 4.7: Lemma 54

is the 1-valent leaf of γ (see Figure 4.7). There are two cases:

(a) $p = p'$. By Lemma 56, p is correct.

(b) $p \neq p'$. Suppose, for contradiction, that p crashes. By Lemma 46, there is a schedule E containing only steps of correct processes (and hence no step of p) such that:

i. $S_0 \cdot E$ is a vertex of Υ^i and all correct processes have decided in $S_0 \cdot E(I^i)$. Since S_0 is 0-valent, all correct processes must have decided 0 in $S_0 \cdot E(I^i)$.

ii. If E is applicable to $S_1(I^i)$ then $S_1 \cdot E$ is a vertex of Υ^i.

Let $S' = S \cdot (p, m, d)$ be the parent of S_1. The state of every process other than p is the same in $S(I^i)$ and $S'(I^i)$. Furthermore, any message addressed to a process other than p that is in the message buffer in $S(I^i)$

is still in the message buffer in $S'(I^i)$. Therefore, since $S_0 = S\cdot(p', m', d')$ and $S_1 = S' \cdot (p', m', d')$, the state of every process other than p is the same in $S_0(I^i)$ and $S_1(I^i)$. In addition, any message addressed to a process other than p that is in the message buffer in $S_0(I^i)$ is also in the message buffer in $S_1(I^i)$. Since E is applicable to $S_0(I^i)$ and does not contain any steps of p, an easy induction on the number of steps in E shows that: (a) E is also applicable to $S_1(I^i)$, and (b) the state of every process other than p is the same in $S_0 \cdot E(I^i)$ and $S_1 \cdot E(I^i)$. By (ii), (a) implies that $S_1 \cdot E$ is a vertex of Υ^i. By (b), all correct processes decide 0 in $S_1 \cdot E(I^i)$. Thus S_1, receives a tag of 0. But S_1 is 1-valent—a contradiction. □

There may be several critical indices and several decision gadgets in the simulation forest. Thus, Lemmata 55 and 57 may identify many correct processes. Our selection rule will choose *one* of these, as the failure detector Ω requires, as follows. It first determines the smallest critical index i. If i is monovalent critical, it selects p_i. If, on the other hand, i is bivalent critical, it chooses the "smallest" decision gadget in Υ^i according to some encoding of gadgets, and selects the corresponding deciding process. It is easy to encode finite graphs as natural numbers. Since a decision gadget is just a finite graph, the selection rule can use any such encoding. The whole method of selecting a correct process is shown in Figure 4.8.

Theorem 58: The algorithm in Figure 4.8 selects a correct process.

PROOF: By Lemma 53, there is a critical index $i, 0 < i \leq n$. If i is monovalent critical, Line 2 returns p_i which, by Lemma 55, is correct. If i is bivalent critical, by Lemma 54, Υ^i contains at least one decision gadget. Let γ be the decision gadget in Υ^i with the smallest encoding. By Lemma 57, the deciding process of γ is correct in F. Thus, Line 3 returns the identity of a process that is correct. □

{*Build and tag simulation forest* Υ *induced by* G}
for $i \leftarrow 0, 1, \ldots, n$:
    $\Upsilon^i \leftarrow$ simulation tree induced by G and I^i
    for every vertex S of Υ^i
        if S has a descendent S' such that a correct process has decided k in $S'(I^i)$
        then add tag k to S

{*Select a process from tagged simulation forest* Υ}
$i \leftarrow$ smallest critical index (1)
if i is monovalent critical **then return** p_i (2)
else return deciding process of the smallest decision gadget in Υ^i (3)

Figure 4.8: Selecting a correct process

4.5.5 The reduction algorithm $T_{\mathcal{D}\to\Omega}$

The selection of a correct process described in Figure 4.8 is not yet the distributed algorithm $T_{\mathcal{D}\to\Omega}$ that we are seeking: it involves an infinite simulation forest and is "centralized". To turn it into a distributed algorithm, we will modify it as follows. Each process will cooperate with other processes to construct ever increasing finite approximations of the simulation forest. Such approximations will eventually contain the decision gadget and the other tagging information necessary to extract the identity of the *same* correct process chosen by the selection method in Figure 4.8.

Note that the selection method in Figure 4.8 involves three stages: The construction of G, a graph representing samples of failure detector values and their temporal relationship; the construction and tagging of the simulation forest induced by G; and, finally, the selection of a correct process using this forest.

Algorithm $T_{\mathcal{D}\to\Omega}$ consists of two components. In the first component, each process repeatedly queries its failure detector module and sends the failure detector values it sees to the other processes. This component enables correct processes to construct ever increasing finite approximations of the *same* G. Since all inter-process communication occurs in this component, we call it the *communication*

{*Build the directed acyclic graph* G_p}
$G_p \leftarrow$ empty graph
repeat forever
  RECEIVE PHASE:
    p receives m
  FAILURE DETECTOR QUERY PHASE:
    $d_p \leftarrow$ query failure detector $\mathcal{D}$
  SEND PHASE:
    if m is of the form (q, G_q, p) **then**
      $G_p \leftarrow G_p \cup G_q$ (1)
    add $[p, d_p]$ to G_p and edges from all other vertices of G_p to $[p, d_p]$ (2)
    $output_p \leftarrow$ *computation component* {*Figure 4.10*} (3)
    p sends (p, G_p, q) to all $q \in \Pi$ (4)

Figure 4.9: Process p's communication component

component of $T_{\mathcal{D}\to\Omega}$.

In the second component, each process repeatedly (a) constructs and tags the simulation forest induced by its current approximation of G, and (b) selects the identity of a process using its current simulation forest. Since this component does not require any communication, we call it the *computation component* of $T_{\mathcal{D}\to\Omega}$.

The communication component

In this component processes cooperate to construct ever increasing approximations of the same G. Let G_p denote p's current approximation of G. Roughly speaking, each process p periodically performs the following two tasks: (i) If p receives G_q for some q, it incorporates this information by replacing G_p with the union of G_p and G_q. (ii) Process p queries its own failure detector module. Let d be the value that it sees and $[p', d']$ be any vertex currently in G_p. Clearly, p saw d after p' saw d'. Thus p adds $[p, d]$ to G_p, with edges from all other vertices of G_p to $[p, d]$. Process p then sends its updated G_p to all other processes. The communication component of $T_{\mathcal{D}\to\Omega}$ for p is shown in Figure 4.9.

Let $G_p(t)$ denote the value of G_p at time t. If p takes a step at time t, $G_p(t)$

denotes the value of G_p *at the end* of that step. The next two lemmata establish certain useful properties of the graphs constructed by the communication component. In reading the proofs of these results it will be useful to keep in mind that in our model the three phases of a step — receive, failure detection query, and send — occur atomically at a single time t.

Lemma 59: Let v be a vertex contained in some local graph during the execution of the communication component. Let $G_p(t)$ be the first graph that contains v. (That is, v is in $G_p(t)$, but not in $G_q(t')$, for any process q and time $t' < t$.) Then

1. $v = [p, d]$, and p saw d at time t.

2. If $u \rightarrow v$ is an edge contained in some local graph during the execution of the communication component then $u \rightarrow v$ is contained in $G_p(t)$.

3. $G_p(t)$ is a subgraph of any graph that contains v.

PROOF: 1. Process p adds v into $G_p(t)$ in Line (1) or (2). In the latter case, the result follows immediately. In the former case, p must have received a message at time t with a graph that contains v. The process that sent that message must have therefore had v in its graph before time t, contradicting the choice of $G_p(t)$ as the first graph to contain v.

2. Consider the earliest time t' when the edge $u \rightarrow v$ was added to some graph, say of process q. By definition of t, $t' \geq t$. If $t' > t$, at time t' process p' receives a message that contains a graph with the edge $u \rightarrow v$. The sender of that message had a graph that contained the edge $u \rightarrow v$ at some time before t', contrary to the choice of t'. Therefore it must be that $t' = t$. Then, by Part (1), $q = p$ and so $u \rightarrow v$ is in $G_p(t)$, as wanted.

3. Suppose, for contradiction, that some graph contains v but is not a supergraph of $G_p(t)$. Choose the first such graph, say, $G_q(t')$. By definition of t, $t' \geq t$. Clearly, $q \neq p$ because p never removes any vertices or edges from its own graph.

Therefore, at time t' process q receives a message with a graph that contains v but is not a supergraph of $G_p(t)$. The sender of that message must have had a graph that contains v but is not a supergraph of $G_p(t)$ before time t', contrary to the choice of $G_q(t')$. $\square$

Recall that we are considering a fixed run of $T_{\mathcal{D}\to\Omega}$, with failure pattern F, and failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$. We now prove that the graphs constructed by the communication component of $T_{\mathcal{D}\to\Omega}$ satisfy certain properties. The reader should note the similarity between the first four and the four properties of the graphs defined in Section 4.5.1.

Lemma 60: For any correct process p and time t:

1. The vertices of $G_p(t)$ are of the form $[p', d']$ where $p' \in \Pi$ and $d' \in \mathcal{R}_{\mathcal{D}}$. If $[p', d']$ is a vertex of $G_p(t)$, then there is a time t' such that $p' \notin F(t')$ and $d' = H_{\mathcal{D}}(p', t')$.
2. If $[q_1, d_1] \to [q_2, d_2]$ is an edge of $G_p(t)$ and $d_1 = H_{\mathcal{D}}(q_1, t_1)$ and $d_2 = H_{\mathcal{D}}(q_2, t_2)$ then $t_1 < t_2$.
3. $G_p(t)$ is transitively closed.
4. There is a time $t' \geq t$ and a failure detector value d such that for all vertices $[p', d']$ of $G_p(t)$, $[p', d'] \to [p, d]$ is an edge of $G_p(t')$.
5. $G_p(t)$ is a subgraph of $G_p(t')$, for all $t' \geq t$.
6. For all correct q, there is a time $t' \geq t$ such that $G_p(t)$ is a subgraph of $G_q(t')$.

PROOF:

Property 1 : Consider the first graph that contains the vertex $[p', d']$. By Lemma 59(1), this graph is $G_{p'}(t')$ for some time t', and p' saw d' at time t'. This means that $p' \notin F(t')$ (otherwise p' would not have taken a step at time t' and would not have seen d'), and $d' = H_{\mathcal{D}}(p', t')$, as wanted.

Property 2 : By Lemma 59(2), $[q_1, d_1] \to [q_2, d_2]$ is an edge of $G_{q_2}(t_2)$. Let t' be the time when q_2 inserted vertex $[q_1, d_1]$ into G_{q_2}. Of course, $t' \leq t_2$. There are two cases:

1. $t' < t_2$. By Lemma 59(1), $[q_1, d_1]$ was not in any graph before time t_1. Thus, $t_1 \leq t'$ and from the hypothesis of this case, $t_1 < t_2$.

2. $t' = t$. Then q_2 received a graph containing $[q_1, d_1]$ at t_2. Let t'' be the time when this graph was sent. Of course, $t'' < t_2$. By Lemma 59(1), $[q_1, d_1]$ was not in any graph before t_1, and therefore $t_1 \leq t''$. Thus, $t_1 < t_2$.

Property 3 : Let $[q_1, d_1] \to \ldots \to [q_k, d_k]$ be a path in $G_p(t)$. We must show that there is an edge $[q_1, d_1] \to [q_k, d_k]$ in $G_p(t)$.

Let t_i be the time when q_i inserted $[q_i, d_i]$ in G_{q_i}, for $1 \leq i \leq k$. By induction on i we show that $[q_1, d_1] \to \ldots \to [q_i, d_i]$ is a path in $G_{q_i}(t_i)$. The basis, $i = 1$, is trivial. For the induction step, suppose that $[q_1, d_1] \to \ldots \to [q_{i-1}, d_{i-1}]$ is a path in $G_{q_{i-1}}(t_{i-1})$. Since $[q_{i-1}, d_{i-1}] \to [q_i, d_i]$ is an edge in $G_p(t)$, by Lemma 59(2), it is also an edge in $G_{q_i}(t_i)$. Since $[q_{i-1}, d_{i-1}]$ is a vertex in $G_{q_i}(t_i)$, by Lemma 59(3), $G_{q_{i-1}}(t_{i-1})$ is a subgraph of $G_{q_i}(t_i)$. In particular, $G_{q_i}(t_i)$ contains the path $[q_1, d_1] \to \ldots \to [q_{i-1}, d_{i-1}]$. Thus, $[q_1, d_1] \to \ldots \to [q_i, d_i]$ is a path in $G_{q_i}(t_i)$, as wanted.

Therefore, the vertices $[q_1, d_1], \ldots, [q_k, d_k]$ are all in $G_{q_k}(t_k)$. At time t_k, q_k adds an edge from every other vertex to $[q_k, d_k]$. Thus, the edge $[q_1, d_1] \to [q_k, d_k]$ is in $G_{q_k}(t_k)$. By Lemma 59(3), $G_{q_k}(t_k)$ is a subgraph of $G_p(t)$ (since the latter contains $[q_k, d_k]$). Therefore, $[q_1, d_1] \to [q_k, d_k]$ is in $G_p(t)$, as wanted.

Property 5 : Once a vertex or edge is added to G_p it is not removed.

Property 4 : Since p is correct, it takes a step at some time t' after t. In the failure detector query phase of this step, p queries its failure detector module and obtains a value, say d. In Line 2 of this step, p adds the vertex $[p, d]$ to G_p and an edge from all other vertices of $G_p(t')$ to $[p, d]$. From Property 5, $G_p(t)$ is a subgraph of $G_p(t')$, hence the result follows.

Property 6 : Since p is correct, it eventually sends $G_p(t)$ to all processes, including q (this occurs in p's first execution of Line 4 after time t). Since q is correct, it eventually receives $G_p(t)$, and then replaces G_q with $G_q \cup G_p(t)$, say at time t'. So, $G_p(t)$ is a subgraph of $G_q(t')$. □

Property 5 of the above lemma allows us to define $G_p^\infty = \bigcup_{t \in \mathcal{T}} G_p(t)$. From Property 6, we get:

Lemma 61: For any correct processes p and q, $G_p^\infty = G_q^\infty$.

PROOF: Let o be any vertex or edge of G_p^∞, i.e., there is a time t at which o is in $G_p(t)$. From Lemma 60 (6), there is a time t' such that $G_p(t)$ is a subgraph of $G_q(t')$. Thus o is in G_q^∞. Thus G_p^∞ is a subgraph of G_q^∞. By a symmetric argument, G_q^∞ is a subgraph of G_p^∞, hence $G_p^\infty = G_q^\infty$. □

Lemma 61 allows us to define the *limit graph* G to be G_p^∞ for any correct process p. The first four properties of Lemma 60 immediately imply:

Lemma 62: The limit graph G satisfies the four properties of the DAG defined in Section 4.5.1.

As before, Υ^i denotes the tagged simulation tree induced by G and initial configuration I^i, and Υ denotes the tagged simulation forest $\{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$.

The computation component

Since the limit graph G has the four properties of the DAG, we can apply the "centralized" selection method of Figure 4.8 to identify a correct process. This method involved:

- Constructing and tagging the infinite simulation forest Υ induced by G.
- Applying a rule to Υ to select a particular correct process p^*.

In the computation component of $T_{\mathcal{D}\to\Omega}$, each p approximates the above method by repeatedly:

- Constructing and tagging the *finite* simulation forest Υ_p induced by G_p, its present finite approximation of G.
- Applying the same rule to Υ_p to select a particular process.

Since the limit of Υ_p over time is Υ, and the information necessary to select p^* is in a finite subgraph of Υ, we can show that *eventually* p will keep selecting the correct process p^*, forever.

Actually, p cannot quite use the tagging method of Figure 4.8: that method requires knowing which processes are correct! Instead, p assigns tag k to a vertex S in Υ_p^i if and only if S has a descendent S' such that p itself has decided k in $S'(I^i)$. If p is correct, this is eventually equivalent to the tagging method of Figure 4.8. If p crashes, we do not care how it tags its forest. Also, p cannot use exactly the same selection method as that of Figure 4.8: its current simulation forest Υ_p may not *yet* have a critical index or contain any decision gadget (although it eventually will!). In that case, p temporizes by just selecting itself. The computation component of $T_{\mathcal{D}\to\Omega}$ is shown in Figure 4.10 (compare it with the selection method of Figure 4.8).

We first show that Υ_p, the simulation forest that p constructs, is indeed an increasingly accurate approximation of Υ (Lemma 63). We then show that the

{*Build and tag simulation forest* Υ_p *induced by* G_p}
for $i \leftarrow 0, 1, \ldots, n$:
    $\Upsilon_p^i \leftarrow$ simulation tree induced by G_p and I^i
    for every vertex S of Υ_p^i
        if S has a descendent S' such that p has decided k in $S'(I^i)$
        then add tag k to S

{*Select a process from tagged simulation forest* Υ_p}
if there is no critical index **then return** p
else
    $i \leftarrow$ smallest critical index (1)
    if i is monovalent critical **then return** p_i (2)
    else if Υ_p^i has no decision gadgets **then return** p
        else return deciding process of the smallest decision gadget in Υ_p^i (3)

Figure 4.10: Process p's computation component

tags that p gives to any vertex S in Υ_p are eventually the same ones that the tagging rule of Figure 4.8 gives to S in Υ (Lemma 64). Let $\Upsilon_p(t)$ denote Υ_p at time t, i.e., $\Upsilon_p(t)$ is the finite simulation forest induced by $G_p(t)$.

Lemma 63: For any correct p and any time t:

1. $\Upsilon_p(t)$ is a subgraph[7] of Υ.

2. $\Upsilon_p(t)$ is a subgraph of $\Upsilon_p(t')$, for all $t' \geq t$.

3. $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t) = \Upsilon$.

PROOF:

Property 1 : Let S be any vertex of tree $\Upsilon_p^i(t)$ (for some i, $0 \leq i \leq n$). From the definition of $\Upsilon_p^i(t)$, S is compatible with some path g of $G_p(t)$ and applicable to I^i. Since $G_p(t)$ is a subgraph of G, g is also a path of G. Thus, S is compatible with G; since it is also applicable to I^i, it is a vertex of Υ^i.

[7]The subgraph and graph equality relations ignore the tags.

Similarly, let $S \rightarrow S'$ be an edge e of $\Upsilon_p^i(t)$. Since S and S' are also vertices of Υ^i, and $S' = S \cdot e$, $S \rightarrow S'$ is also an edge of Υ^i.

Property 2 : Follows from Lemma 60 (5).

Property 3 : We first show that Υ is a subgraph of $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$. Let S be any vertex of any tree Υ^i of Υ. From the definition of Υ^i, S is compatible with some finite path g of G and is applicable to I^i. Since $G = \bigcup_{t \in \mathcal{T}} G_p(t)$ and g is a finite path of G, there is a time t such that g is also a path of $G_p(t)$. Since S is compatible with g of $G_p(t)$ and is applicable to I^i, S is a vertex of $\Upsilon_p^i(t)$.

Let $S \rightarrow S'$ be any edge e of Υ^i. By the argument above, there is a time t after which both S and S' are vertices of Υ_p^i. Since $S' = S \cdot e$, after time t the edge e is also in Υ_p^i. Thus, every vertex and every edge of Υ is also in $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$, i.e., Υ is a subgraph of $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$.

By Property 1, $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t) = \Upsilon$. □

Lemma 64: Let p be any correct process, and S be any vertex of Υ_p. There is a time after which the tags of S in Υ_p are the same as the tags of S in Υ.

PROOF: Suppose that at some time t, p assigns tag k to vertex S of tree Υ_p^i. This means that S has a descendent S' in $\Upsilon_p^i(t)$ such that p has decided k in $S'(I^i)$. By Lemma 63(1), S' is also a descendent of S in Υ^i, and since p is correct, S has tag k in Υ^i as well.

Conversely, suppose a vertex S of a tree Υ^i of Υ has tag k. We show that, eventually, p also assigns tag k to S in Υ_p^i. Since S has tag k in Υ^i, S has a descendent S' in Υ^i such that some correct process has decided k in $S'(I^i)$ (cf. tagging rule in Figure 4.8). By Lemma 46(1), there is a descendent S'' of S' in Υ^i, such that *all* correct processes, including p, have decided in $S''(I^i)$. By Lemma 41, $S''(I^i)$ is a configuration of a partial run of $\mathit{Consensus}_{\mathcal{D}}$. By the

Agreement property of Consensus, p must have decided k in $S''(I^i)$. Consider the path that starts from the root of Υ^i and goes to vertex S and then to S''. By Lemma 63(3), there is a time t after which this path is also in Υ^i_p. Therefore, when p executes the tagging rule of Figure 4.10 after time t, p assigns tag k to S in Υ^i_p (because p has decided k in $S''(I^i)$, and S'' is a descendent of S in Υ^i_p). $\square$

Recall that p^* is the correct process obtained by applying the selection rule of Figure 4.8 to the infinite simulation forest Υ. We now show that there is a time after which any correct p always selects p^* when it applies the corresponding selection rule of Figure 4.10 to its own finite approximation of the simulation forest Υ_p. Roughly speaking, the reason is as follows. By Lemma 64, there is a time t after which the tags of all the roots in p's forest Υ_p are the same as in the infinite forest Υ. Since these tags determine the sets of monovalent and bivalent critical indices, after time t these sets according to p are the same as in Υ. Let i be the minimum critical index in these sets, and consider the situation after time t. If i is monovalent critical, the selection rule of Figure 4.10 selects p_i, which is what p^* is in this case. If i is bivalent critical, then p selects the deciding process of its current minimum decision gadget of Υ^i_p (if it has one). This case is examined below.

Let γ^* be the minimum decision gadget of Υ^i (so, p^* is the deciding process of γ^*). For a while, γ^* may not be the minimum decision gadget of Υ^i_p. This may be because γ^* (and its tags) is not yet in Υ^i_p. However, by Lemmata 63(3) and 64, γ^* (including its tags) will eventually be in Υ^i_p. Alternatively, it may be because Υ^i_p contains a subgraph γ whose encoding is smaller than γ^*'s, and for a while γ looks like a decision gadget according to its *present* tags. However, by Lemma 64, p will eventually determine *all* the tags of γ, and discover that γ is not really a decision gadget. Since there are only *finitely* many graphs whose encoding is smaller than γ^*'s, p will eventually discard all the "fake" decision gadgets (like γ) that are smaller than γ^*, and then select γ^* as its minimum decision gadget. After that time, p always selects the deciding process of γ^* — which is precisely p^*, in this case.

Theorem 65: For all correct processes p, there is a time after which $output_p = p^*$, forever.

PROOF: Let i^* denote the critical index selected by Line 1 of Figure 4.8 applied to Υ. By Lemma 64, there is a time t_{init} after which every root of Υ_p has the same tags as the corresponding root of Υ. Thus after time t_{init}, p always sets $i = i^*$ in Line 1 of Figure 4.10. We now show that there is a time after which the computation component of p (Figure 4.10) always return p^*. There are two cases:

1. i^* is monovalent critical. In this case, p^* is process p_{i^*} (by Line 2 of the selection rule Figure 4.8). Similarly, after time t_{init}: (a) p always sets i to i^* (Line 1 of Figure 4.10); (b) p always returns p_{i^*} (Line 2 of Figure 4.10).

2. i^* is bivalent critical. Let γ^* denote the smallest decision gadget of Υ^{i^*}. In this case, p^* is the deciding process of γ^*. Since γ^* is a finite subgraph of Υ^{i^*}, by Lemma 63(3), there is a time after which γ^* is also a subgraph of Υ^i_p. By Lemma 64, there is a time t_{γ^*} after which all the (finitely many) vertices of γ^* receive the same tags in Υ^{i^*} and $\Upsilon^{i^*}_p$. Thus after time t_{γ^*}, γ^* is a also decision gadget of $\Upsilon^{i^*}_p$.

   Since each graph is encoded as a unique natural number, there are finitely many graphs with a smaller encoding than γ^*. Let $\mathcal{G}$ denote the set of graphs with a smaller encoding than γ^*, and γ be any graph in $\mathcal{G}$. We show that there is a time after which γ is not a decision gadget of $\Upsilon^{i^*}_p$. There are two cases:

   (a) γ is not a subgraph of Υ^{i^*}. In this case, by Lemma 63(1), γ is never a subgraph of $\Upsilon^{i^*}_p$.

(b) γ is a subgraph of Υ^{i^*}. Since γ^* is the smallest decision gadget of Υ^{i^*} and γ is smaller than γ^*, γ is not a decision gadget of Υ^{i^*}. By Lemma 64, there is a time t_γ after which all the (finitely many) vertices of γ have the same tags in Υ^{i^*} and $\Upsilon_p^{i^*}$. Thus after time t_γ, γ is not a decision gadget of Υ_p^i.

Since $\mathcal{G}$ is finite, there is a time $t_{\mathcal{G}}$ after which no graph in $\mathcal{G}$ is a decision gadget of Υ_p^i.

Consider the process that is returned by the computation component of p (Figure 4.10) at any time $t > \max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$. Since $t > t_{init}$, p always sets i to i^* in Line 1. Since $t > t_{\gamma^*}$, γ^* is a decision gadget of $\Upsilon_p^i(t)$. Finally, since $t > t_{\mathcal{G}}$, γ^* is the smallest decision gadget of $\Upsilon_p^i(t)$. Thus, since i^* is bivalent, at any time after $\max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$, Line 3 of Figure 4.10 returns the deciding process of γ^*. Therefore, after time $\max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$, the computation component of p always returns p^*.

From the above, there is a time after which p sets $output_p \leftarrow p^*$, forever, in Line 3 of Figure 4.9. □

We now have all the pieces needed to prove our main result, Theorem 36 in Section 4.4:

Theorem 36: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \Omega$.

PROOF: Consider the execution of algorithm $T_{\mathcal{D}\to\Omega}$ in any environment $\mathcal{E}$. By Theorem 65, there is a time after which all correct processes set $output_p = p^*$, forever. By Theorem 58, p^* is a correct process. Thus, $T_{\mathcal{D}\to\Omega}$ is a reduction algorithm that transforms $\mathcal{D}$ into Ω. In other words, Ω is reducible to $\mathcal{D}$. □

4.6 Discussion

4.6.1 Granularity of atomic actions

Our model incorporates very strong assumptions about the atomicity of steps. First, the three phases of each step are assumed to occur indivisibly, and at a single time. In particular, the failure of a process cannot happen in the "middle of a step". This allows us to associate a single time t with a step and think of the step as occurring at that time. Second, in the send phase of a step a message is sent to *all* processes. Given that the entire step is indivisible, this means that either all or none of the correct processes eventually receive the message sent in a step. Finally, no two steps can occur at the same time.[8] These assumptions are convenient because they make the formal model simpler to describe. Also, they are consistent with those made in the model of [FLP85] that provided the impetus for this work.

On the other hand, in Chapter 3 a model with weaker properties is used. There, the three phases of a step need not occur indivisibly, and may occur at different times. Even within the send phase, the messages sent to the different processes may be sent at different times. Thus, a failure may occur in the middle of the send phase, resulting in some correct processes eventually receiving the messages sent to them in that step while others never do. Also, actions of *different* processes may take place simultaneously, subject to the restriction that a message can only be received strictly *after* it was sent. Since Chapter 3 is mainly concerned with showing how to use various types of failure detectors to achieve Consensus, the use of a weaker model strengthens the results. (In fact, the negative results of the Appendix hold even in the model of this Chapter, with the stronger atomicity assumptions.)

The question naturally arises whether our result also applies to this weaker

[8] This is reflected in our formal model by the fact that the list of times in a run (which indicate when the events in the run's schedule occur) is *increasing*.

model. In other words, if a failure detector $\mathcal{D}$ can be used to solve Consensus in the weak model, is it true that we can transform $\mathcal{D}$ to $\diamond\mathcal{W}$ *in that model*? It turns out that the answer is affirmative. To see this, first note that if $\mathcal{D}$ solves Consensus in the weak model then it surely solves Consensus in the strong model. By our result, $\mathcal{D}$ can be transformed to $\diamond\mathcal{W}$ in the strong model. It remains to show that $\mathcal{D}$ can be transformed to $\diamond\mathcal{W}$ in the weak model. This is not obvious, since it is conceivable that the extra properties of the strong model are crucial in the transformation of $\mathcal{D}$ to $\diamond\mathcal{W}$. Fortunately, the transformation presented in this chapter actually works even in the weak model!

To see this, it is sufficient to make sure that the communication component of the transformation (cf. Figure 4.9 in Section 4.5.5) constructs graphs that satisfy the properties listed in Lemma 60, *even* if we run it in the weak model. It is not difficult to verify that this is indeed so. The proof is virtually the same, except for the fact that we must distinguish the time t in which a process p queries its failure detector and the time t' in which p adds the value it saw into G_p. In our proof we assume that $t = t'$; in the weak model we would have $t \leq t'$. Similar comments apply to all actions within a step that are no longer assumed to occur at the same instant of time. These changes make the proofs slightly more cumbersome, since we must introduce notation for all the different times in which relevant actions within a step take place, but the reasoning remains essentially the same.[9]

Thus, our result is not merely a fortuitous consequence of some whimsical choice of model. We view the robustness of the result across different models of asynchrony as further testimony to the significance of the failure detector $\diamond\mathcal{W}$.

[9]Another problem that must be confronted is that in the proofs of Lemmata 59 and 60 we often refer to the "first graph" in which a vertex or edge is present. In the strong model there is no difficulty with this, since processes cannot execute steps simultaneously. In the weak model, we have to justify that it makes sense to speak of the "first" graph to contain a vertex or edge, in spite of the fact that certain actions can be executed at the same time. The fact that a message can be received only *after* it was sent is needed here.

4.6.2 Weak Consensus

[FLP85] actually showed that even the *Weak* Consensus problem cannot be solved (deterministically) in an asynchronous system. Weak Consensus is like Consensus except that the validity property is replaced by the following, weaker, property:

Non-triviality: There is a run of the protocol in which correct processes decide 0, and a run in which correct processes decide 1.

Unlike validity, this property does not explicitly prescribe conditions under which the correct processes must decide 0 or 1 — it merely states that it is possible for them to reach each of these decisions. It is natural to ask whether our result holds for this weaker problem as well. That is, we would like to know if the following holds:

Theorem: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve *Weak* Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \Omega$.

Under the above definition of non-triviality, this is not quite right. But as we shall argue, the problem really lies with the definition! Under a slightly stronger definition, which is more appropriate for our model that incorporates failure detectors, the theorem is actually true.

Intuitively, the problem with the above definition of non-triviality in our model of failure detectors is that it is possible for the decision of correct processes to depend entirely on the the values returned by the failure detector. Consider, for example, a failure detector $\mathcal{D}$ so that for each failure pattern F, $\mathcal{D}(F) = \{H_0, H_1\}$, where for all processes p and times t, and for all $i \in \{0,1\}$, $H_i(p,t) = i$. In other words, in any given run, this failure detector returns the same binary value to all processes at all times, independent of the run's failure pattern. It is trivial to use this failure detector to solve Weak Consensus: A process merely queries its failure detector and decides the value returned! It is easy to see that $\mathcal{D} \not\succeq_{\mathcal{E}} \Omega$, for any environment $\mathcal{E}$: $\mathcal{D}$ provides absolutely no information about which processes are correct or faulty.[10]

At this point, the reader may justifiably object that $\mathcal{D}$ is "cheating" — it is really not a failure detector, but a mechanism that non-deterministically chooses the decision value. One possible way of fixing this problem would be to make our definition of failure detector less general than it presently is. We could then try to prove the theorem for this restricted definition of failure detectors. This approach, however, is fraught with the danger of restricting the definition too much and ruling out legitimate failure detectors in addition to bogus ones, like $\mathcal{D}$. Intuitively, the failure detector is supposed to provide some information about faulty processes. As this information may be encoded in a complex way, we should not arbitrarily rule out such encodings because, in doing so, we may be inadvertently ruling out useful failure detectors.

Instead of modifying our definition of failure detector, we strengthen the non-triviality property to require that the failure detector values seen by the processes do not, by themselves, determine the decision value. To formalise this, let R be a run of a Consensus algorithm, and $(p_1, m_1, d_1), (p_2, m_2, d_2), \ldots$, be the schedule of R. We denote by $\mathbf{fd}(R)$ the sequence $[p_1, d_1][p_2, d_2]\ldots$, i.e., the sequence of failure detector values seen by the processes in R. Consider the relation $\equiv$ on runs, where $R \equiv R'$ if and only if $\mathbf{fd}(R) = \mathbf{fd}(R')$. It is immediate that $\equiv$ is an equivalence relation. We now redefine the non-triviality property in our model (where processes have access to failure detectors) as follows:

Non-triviality: In every equivalence class of the relation $\equiv$, there is a run of the protocol in which correct processes decide 0, and a run in which correct processes decide 1.

This captures the idea that the decision value cannot be ascertained merely on the basis of the failure detector values seen by the processes. It must also depend on

[10] In fact, $\mathcal{D}$ cannot be used to solve Consensus.

other aspects of the run (such as the initial values, the particular messages sent, or other features).

If we define Weak Consensus using *this* version of non-triviality, then the Theorem stated above is, in fact, true. We briefly sketch the modifications of our proof needed to obtain this strengthening of Theorem 36. The only use of the validity property is in the proof of Lemma 52 which states that the root of Υ^0 is 0-valent and the root of Υ^n is 1-valent. This, in turn, is used in the proof of Lemma 53, which states that a critical index exists.

To prove the stronger theorem, we concentrate on the forest induced by *all* initial configurations — not just $I^0, \dots, I^n$. Thus, the forest now will have 2^n trees, rather than only $n+1$. Consider the n initial values of processes in an initial configuration as an n-bit vector, and fix any n-bit Gray code.[11] Let $I^0, \dots, I^{2^n-1}$ be the initial configurations listed in the order specified by the Gray code, and Υ^i be the tree $\Upsilon_G^{I^i}$, for all $i \in \{0, \dots, 2^n - 1\}$. We use the same definition for a critical index as we had before: Index $i \in \{0, \dots, 2^n - 1\}$ is *critical* if the root of Υ^i is bivalent or the root of Υ^i is 1-valent while the root of Υ^{i-1} is 0-valent. The only difference is that we now take subtraction to be modulo 2^n, so that when $i = 0$, $i - 1 = -1 = 2^n - 1$. We can now prove an analogue to Lemma 53.

Lemma: There is a critical index i, $0 \leq i \leq 2^n - 1$.

PROOF: First, we claim that that the forest contains both nodes tagged 0 and nodes tagged 1. To see this, let S be a node in some tree of the forest. By Lemma 47, S has a tag; without loss of generality, assume that S has tag 0. Consider an infinite path that extends S. By Lemma 42 and the fact that S is tagged 0, there is a run R of the Weak Consensus algorithm in which a correct process decides 0. By non-triviality, there is a run $R' \equiv R$, so that correct processes decide 1 in R'. Let S' be the infinite schedule and I^ℓ be the initial configuration of

[11] An n-bit Gray code is a sequence of all possible n-bit vectors where successive vectors, as well as the first and last vectors, differ only in the value of one position. It is well-known that such codes exist for all $n \geq 1$.

run R'. Using the definition of the $\equiv$ relation and the construction of the induced forest, it is easy to show that every finite prefix of S' is a node of Υ^ℓ. Since correct processes decide 1 in R', all these nodes are tagged 1.

Since there are both nodes tagged 0 and nodes tagged 1, by Lemma 49, there are both roots tagged 0 and roots tagged 1. If the root of some Υ^i is tagged both 0 and 1, it is bivalent and we are done. Otherwise, the roots of all trees are monovalent, and there are both 0- and 1-valent roots. Thus, there exist $0 \leq i, j \leq 2^n - 1$ so that the root of Υ^i is 0-valent and the root of Υ^j is 1-valent. By considering the sequence $\Upsilon^i, \Upsilon^{i+1}, \ldots, \Upsilon^j$, (where addition is modulo 2^n) it is easy to see that the root of some Υ^k, $k \neq i$, that appears in that sequence is 1-valent, while the root of Υ^{k-1} is 0-valent. By definition, k is a critical index. $\square$

The rest of the proof remains unchanged.

4.6.3 Failure detectors with infinite range of output values

The failure detectors in [RB91] and Chapter 3 only output lists of processes suspected to have crashed. Since the set of processes is finite, the range of possible output values of these failure detectors is also finite. In this chapter our model allows for failure detectors with arbitrary ranges of output values, including the possibility of infinite ranges! We illustrate the significance of this generality by describing a natural class of failure detectors whose range of output values is infinite (though each value output is finite).

Example: One apparent weakness with our formulation of failure detection is that a brief change in the value output by a failure detector module may go unnoticed. For example, process p's module of the given failure detector, $\mathcal{D}$, may output d_1 at time t_1, d_2 at a later time t_2 and d_1 again at time t_3 after t_2. If due to the asynchrony of the system p does not take a step between time t_1 and t_3, p may never notice that its failure detector module briefly output d_2. A natural way of overcoming this problem is to replace $\mathcal{D}$ with failure detector $\mathcal{D}'$ that has the following property: $\mathcal{D}'$ maintains the same list of suspects as $\mathcal{D}$ but when queried, $\mathcal{D}'$ returns the entire history of its list of suspects up to the present time. In this manner, correct processes are guaranteed to notice every change in $\mathcal{D}'$'s list of suspects. As the system continues executing, the values output by $\mathcal{D}'$ grow in size. This means that $\mathcal{D}'$ has an infinite range of output values.

However, since $\mathcal{D}$ is a function of F, the failure pattern encountered, $\mathcal{D}'$ is also a function of F, and can be described by our model. Thus, the result in this chapter applies to $\mathcal{D}'$, a natural failure detector with infinite range of output values.

4.6.4 Open problems

In our model, each process "polls" its failure detector module each time it takes a step. One can conceive of an alternative model in which the failure detector module can "interrupt" a process; for example, it could issue an interrupt every time there is a change in its list of suspects. We do not know if our results apply to such a model.

In our model, the *specification* of a failure detector, (i.e., its allowable outputs) depends only on *failure patterns.* In other words, the set of failure detector outputs that are allowed for a given execution depends only on the identity of the processes that crash and on the timing of these crashes in that execution.[12] We have not studied more general failure detectors whose specification also depends on other aspects of executions, such as the timing and content of messages, the state of the processes, etc. A major problem that needs to be resolved before this issue can be pursued is to identify which aspects of an execution it would be "fair" to allow the failure detector to depend on. Clearly, the failure detector cannot be allowed to depend on the entire state of the system, for then it would be too powerful and, therefore, of no practical value, as it would be impossible to implement it.

[12] Recall that we allow the *implementation* of a failure detector to use any other aspect of the given execution, such as the local time and/or order in which messages are sent and received.

In this thesis we have focused on failure detectors for systems with *crash* failures only. Extending our results to other types of failures, such as *omission* (cf. [MSF87, Had84,PT86]) and *arbitrary* failures (cf. [PSL80]), remains a goal for future work.

We showed that $\Diamond\mathcal{W}$ is weaker than any failure detector that can be used to solve Consensus. Since $\Diamond\mathcal{W}$ can be used to solve Consensus in any environment in which $n > 2f$, it is the weakest failure detector for solving Consensus in such an environment. In the Appendix we show that $\Diamond\mathcal{W}$ cannot be used to solve Consensus in environments in which $n \leq 2f$. It is still not known (a) whether a weakest failure detector for solving Consensus in such environments exists, and (b) provided it does, what its properties are.

Chapter 5

Related work

5.1 Partial synchrony

Fischer, Lynch and Paterson showed that Consensus cannot be solved in an asynchronous system subject to crash failures [FLP85]. The fundamental reason why Consensus cannot be solved in completely asynchronous systems is the fact that, in such systems, it is impossible to reliably distinguish a process that has crashed from one that is merely very slow. In other words, Consensus is unsolvable because accurate failure detection is impossible. On the other hand, it is well-known that Consensus is solvable (deterministically) in completely synchronous systems — that is, systems where clocks are perfectly synchronised, all processes take steps at the same rate and each message arrives at its destination a fixed and known amount of time after it is sent. In such a system we can use timeouts to implement a "perfect" failure detector — i.e., one in which no process is ever wrongly suspected, and every faulty process is eventually suspected. Thus, the ability to solve Consensus in a given system is intimately related to the failure detection capabilities of that system. This realisation led us to augment the asynchronous model of computation with unreliable failure detectors as described in this thesis.

A different tack on circumventing the unsolvability of Consensus is pursued in [DDS87] and [DLS88]. The approach of those papers is based on the observation that between the completely synchronous and completely asynchronous models of distributed systems there lie a variety of intermediate "partially synchronous" models.

In particular, [DDS87] defines a space of 32 models by considering five key parameters, each of which admits a "favourable" and an "unfavourable" setting. For instance, one of the parameters is whether the maximum message delay is bounded and known (favourable setting) or unbounded (unfavourable setting). Each of the 32 models corresponds to a particular setting of the 5 parameters. [DDS87] identifies four "minimal" models in which Consensus is solvable. These are minimal in the sense that the weakening of any parameter from favourable to unfavourable would yield a model of partial synchrony where Consensus is unsolvable. Thus, within the space of the models considered, [DDS87] delineates precisely the boundary between solvability and unsolvability of Consensus, and provides an answer to the question "What is the least amount of synchrony sufficient to solve Consensus?".

[DLS88] considers the following two models of partial synchrony. The first model assumes that there are bounds on relative process speeds and on message transmission times, but these bounds are not known. The second model assumes that these bounds are known, but they hold only after some unknown time.

In each one of these two models (with crash failures), it is easy to implement an Eventually Perfect Failure Detector $\Diamond\mathcal{P}$. In fact, we can implement $\Diamond\mathcal{P}$ in an even weaker model of partial synchrony: one in which there are bounds on message transmission times and relative process speeds, but these bounds are not known *and* they hold only after some unknown time. Since $\Diamond\mathcal{P}$ is stronger than $\Diamond\mathcal{W}$, by Corollaries 20 and 32, this implementation immediately gives Consensus and Atomic Broadcast solutions for this model of partial synchrony and, a fortiori, for the two models of [DLS88]. The implementation of $\Diamond\mathcal{P}$ is given in Figure 5.1, and proven below.

Every process p executes the following:

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

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

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

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

Figure 5.1: A time-out based implementation of $\Diamond\mathcal{P}$ in some models of partial synchrony.

Each process p periodically sends a "p-is-alive" message to all the processes. If p does not receive a "q-is-alive" message from some process q for $\Delta_p(q)$ units of time, p adds q to its list of suspects. If p receives "q-is-alive" from some process q that it currently suspects, p knows that its previous time-out on q was premature. In this case, p removes q from its list of suspects and increases the length of the time-out.

Theorem 66: Consider a system in which, after some time t, some bounds on relative process speeds and on message transmission times hold (we do not assume that t or the value of these bounds are known). The algorithm in Figure 5.1 implements an Eventually Perfect Failure Detector $\diamond\mathcal{P}$ in this system.

PROOF: (*sketch*) We first show that strong completeness holds, i.e., eventually every process that crashes is permanently suspected by every correct process. Suppose a process q crashes. Clearly, q eventually stops sending "q-is-alive" messages, and there is a time after which no correct process receives such a message. Thus, there is a time t' after which: (1) all correct processes time-out on q (Task 2), and (2) they do not receive any message from q after this time-out. From the algorithm, it is clear that after time t', all correct processes will permanently suspect q. Thus, strong completeness is satisfied.

We now show that eventual strong accuracy is satisfied. That is, for any correct processes p and q, there is a time after which p will not suspect q. There are two possible cases:

1. Process p times-out on q finitely often (in Task 2). Since q is correct and keeps sending "q-is-alive" messages forever, eventually p receives one such message after its last time-out on q. At this point, q is permanently removed from p's list of suspects (Task 3).

2. Process p times-out on q infinitely often (in Task 2). Note that p times-out on q (and so p adds q to $output_p$) only if q is not already in $output_p$. Thus,

q is added to and removed from $output_p$ infinitely often. Process q is only removed from $output_p$ in Task 3, and every time this occurs the time-out period $\Delta_p(q)$ is increased. Since this occurs infinitely often, $\Delta_p(q)$ grows unbounded. Thus, eventually (1) the bounds on relative process speeds and on message transmission times hold, and (2) $\Delta_p(q)$ is larger than the *correct* time-out based on these bounds. After this point, p cannot time-out on q any more—a contradiction to our assumption that p times-out on q infinitely often. Thus Case 2 cannot occur. □

Thus, failure detectors can be viewed as a more abstract and modular way of incorporating partial synchrony assumptions into the model of computation. Instead of focusing on the *operational features* of partial synchrony (such as the five parameters considered in [DDS87]), we can consider the *axiomatic properties* that failure detectors must have in order to solve Consensus. The problem of implementing a given failure detector in a specific model of partial synchrony becomes a separate issue; this separation affords greater modularity.

Studying failure detectors rather than various models of partial synchrony has other advantages as well. By showing that Consensus is solvable using some specific failure detector we thereby show that Consensus is solvable in *all* systems in which that failure detector can be implemented. An algorithm that relies on the axiomatic properties of a given failure detector is more general, more modular, and simpler to understand than one that relies directly on specific operational features of partial synchrony (that can be used to implement the given failure detector).

From this more abstract point of view, the question "What is the least amount of synchrony sufficient to solve Consensus?" translates to "What is the weakest failure detector sufficient to solve Consensus?". In contrast to [DDS87], which identified a *set* of minim*al* models of partial synchrony in which Consensus is solvable, we are able to exhibit a *single* minim*um* failure detector that can be used to solve Consensus. The technical device that made this possible is the notion of *reduction* between failure detectors. We suspect that a corresponding notion of reduction between models of partial synchrony, although possible, would be more complex. This is because there are models which are not comparable in general (in the sense that there are tasks that are possible in one but not in the other and vice versa), although they are comparable *as far as failure detection is concerned* — which is all that matters for solving Consensus! In this connection, it is useful to recall our earlier observation, that the same failure detector can be implemented in different (indeed, incomparable) models of partial synchrony.

5.2 The application of failure detection in shared memory systems

Loui and Abu-Amara showed that in an asynchronous shared memory system with atomic read/write registers, Consensus cannot be solved even if at most one process may crash [LA87]. This raises the following natural question: can we circumvent this impossibility result using unreliable failure detectors? In a recent work, Lo shows that this is indeed possible [Lo93]. In particular, he shows that using a Strong Failure Detector and atomic registers, one can solve Consensus for any number of failures. He also shows that for systems with a majority of correct processes, it is sufficient to use an Eventually Strong Failure Detector and atomic registers.

5.3 The Isis toolkit

Isis is a programming toolkit for building fault-tolerant distributed systems [BJ87, BCJ+90]. Although Isis employs the asynchronous model of distributed computing, it also provides several powerful primitives, including Atomic Broadcast. Roughly speaking, Isis has the following internal architecture. The lowest layer of Isis can be modeled as an asynchronous system and a failure detector. Higher layers of Isis use the lowest layer to implement several primitives including Atomic Broadcast. In this section, we compare our model with the lowest layer of the Isis system.

With our approach, even if a correct process p is repeatedly suspected to have crashed by the other processes, it is still required to behave like every other correct process in the system. For example, with Atomic Broadcast, p is still required to A-deliver the same messages, in the same order, as all the other correct processes. Furthermore, p is not prevented from A-broadcasting messages, and these messages must eventually be A-delivered by *all* correct processes (including those processes whose local failure detector modules permanently suspect p to have crashed). In summary, application programs that use unreliable failure detection are aware that the information they get from the failure detector may be incorrect: they only take this information as an imperfect "hint" about which processes have really crashed. Furthermore, processes are never "discriminated against" if they are falsely suspected to have crashed.

Isis takes an alternative approach based on the assumption that failure detectors rarely make mistakes [RB91]. In those cases in which a correct process p is falsely suspected by the failure detector, p is effectively forced "to crash" (via a *group membership* protocol that removes p from all the groups that it belongs to). An application using such a failure detector cannot distinguish between a faulty process that really crashed, and a correct one that was forced to do so. Essentially, the Isis failure detector forces the system to conform to its view. From the application's point of view, this failure detector looks "perfect": it never makes visible mistakes.

For the Isis approach to work, the low-level time-outs used to detect crashes must be set very conservatively: Premature time-outs are costly (each results in the removal of a process), and too many of them can lead to system shutdown.[1] In contrast, with our approach, premature time-outs (e.g., failure detector mistakes) are not so deleterious: they can only *delay* an application. In other words, pre-

[1] For example, the time-out period in the current version of Isis is greater than 10 seconds.

mature time-outs can affect the *liveness* but not the *safety* of an application. For example, consider the Atomic Broadcast algorithm that uses $\diamond\mathcal{W}$. If the failure detector "malfunctions", some messages may be delayed, but no message is ever delivered out of order, and no correct process is removed. If the failure detector stops malfunctioning, outstanding messages are eventually delivered. Thus, we can set time-out periods more aggressively than Isis: in practice, we would set our failure detector time-out periods closer to the average case, while Isis must set time-outs to the worst-case.

As we have seen, the approach taken in the lowest layer of the Isis system, is fundamentally different from our approach. However, both approaches achieve the same result: by augmenting the asynchronous model of computation with a failure detection mechanism, they make it practical. Thus it would be interesting to study whether the higher layers of Isis can be built based on our approach. This is left as a subject for future research.

5.4 Other work

Several works in fault-tolerant computing used time-outs primarily or exclusively for the purpose of failure detection. An example of this approach is given by an algorithm in [ADLS91], which, as pointed out by the authors, "can be viewed as an asynchronous algorithm that uses a fault detection (e.g., timeout) mechanism."

Appendix A

A hierarchy of failure detectors and bounds on fault-tolerance

In the preceding chapters, we introduced the concept of unreliable failure detectors that could make mistakes, and showed how to use them to solve Consensus despite such mistakes. Informally, a mistake occurs when a correct process is erroneously added to the list of processes that are suspected to have crashed. In this Appendix, we formalise this concept and study a related property that we call *repentance*. Informally, if a process p learns that its failure detector module $\mathcal{D}_p$ made a mistake, repentance requires $\mathcal{D}_p$ to take corrective action. Based on mistakes and repentance, we define a hierarchy of failure detector specifications that will be used to unify some of our results, and to refine the lower bound on fault-tolerance given in Section 3.5.3. This infinite hierarchy consists of a continuum of repentant failure detectors ordered by the maximum number of mistakes that each one can make.

A.1 Mistakes and repentance

We now define a *mistake*. Let $R = \langle F, H, I, S, T \rangle$ be any run using a failure detector $\mathcal{D}$. $\mathcal{D}$ makes a *mistake in* R *at time* t *on process* p *about process* q if at time t, p begins to suspect that q has crashed even though $q \notin F(t)$. Formally:

$$[q \notin F(t), q \in H(p,t)] \text{ and } [\exists t' < t, \forall t'', t' \leq t'' < t : q \notin H(p,t'')]$$

Such a mistake is denoted by the tuple $\langle R, p, q, t\rangle$. The set of mistakes made by $\mathcal{D}$ in R is denoted by $M(R)$.

Note that only the erroneous *addition* of q into $\mathcal{D}_p$ is counted as a mistake on p. The continuous *retention* of q into $\mathcal{D}_p$ does not count as additional mistakes. Thus, a failure detector can make multiple mistakes on a process p about another process q only by repeatedly adding and then removing q from the set $\mathcal{D}_p$. In practice, mistakes are caused by premature time-outs.

We define the following four types of accuracy properties for a failure detector $\mathcal{D}$ based on the mistakes made by $\mathcal{D}$:

- *Strongly $k-$mistaken:* $\mathcal{D}$ makes at most k mistakes. Formally, $\mathcal{D}$ is strongly $k-$mistaken if:

$$\forall R \text{ using } \mathcal{D} : |M(R)| \leq k$$

- *Weakly $k-$mistaken:* There is a correct process p such that $\mathcal{D}$ makes at most k mistakes about p. Formally, $\mathcal{D}$ is weakly $k-$mistaken if:

$$\forall R = \langle F, H, I, S, T\rangle \text{ using } \mathcal{D}, \exists p \in \mathit{correct}(F) :$$
$$|\{\langle R, q, p, t\rangle : \langle R, q, p, t\rangle \in M(R)\}| \leq k$$

- *Strongly finitely mistaken:* $\mathcal{D}$ makes a finite number of mistakes. Formally, $\mathcal{D}$ is strongly finitely mistaken if:

$$\forall R \text{ using } \mathcal{D} : M(R) \text{ is finite.}$$

  In this case, it is clear that there is a time t after which $\mathcal{D}$ stops making mistakes.

- *Weakly finitely mistaken:* There is a correct process p such that $\mathcal{D}$ makes a finite number of mistakes about p. Formally, $\mathcal{D}$ is weakly finitely mistaken if:

$$\forall R = \langle F, H, I, S, T\rangle \text{ using } \mathcal{D}, \exists p \in \mathit{correct}(F) : \\ \{\langle R, q, p, t\rangle : \langle R, q, p, t\rangle \in M(R)\} \text{ is finite.}$$

  In this case, there is a time t after which $\mathcal{D}$ stops making mistakes about p.

For most values of k, the properties mentioned above are not powerful enough to be useful. For example, suppose every process permanently suspects every other process. In this case, the failure detector makes at most $(n-1)^2$ mistakes, but it is clearly useless since it does not provide any information.

The core of this problem is that such failure detectors are not forced to reverse a mistake, even when a mistake becomes "obvious" (say, after a process q replies to an inquiry that was sent to q after q was suspected to have crashed). However, we can impose a natural requirement to circumvent this problem. Consider the following scenario. The failure detector module at process p erroneously adds q to $\mathcal{D}_p$ at time t. Subsequently, p sends a message to q and receives a reply. This reply is a proof that q had not crashed at time t. Thus, p *knows* that its failure detector module made a mistake about q. It is reasonable to require that, given such irrefutable evidence of a mistake, the failure detector module at p takes the corrective action of removing q from $\mathcal{D}_p$. In general, we can require the following property:

- *Repentance:* If a correct process p eventually *knows* that $q \notin F(t)$, then at some time after t, $q \notin \mathcal{D}_p$. Formally, $\mathcal{D}$ is *repentant* if:

$$\forall R = \langle F, H, I, S, T\rangle \text{ using } \mathcal{D}, \forall t, \forall p, q \in \Pi : \\ [\exists t' : (R, t') \models K_p(q \notin F(t))] \Rightarrow [\exists t'' \geq t : q \notin H(p, t'')]$$

The knowledge theoretic operator K_p can be defined formally [HM90]. Informally $(R,t) \models \phi$ iff in run R at time t, predicate ϕ holds. We say $(R,t) \sim_p (R',t')$ iff the run R at time t and the run R' at time t' are indistinguishable to p. Finally, $(R,t) \models K_p(\phi) \iff [\forall (R',t') \sim_p (R,t) : (R',t') \models \phi]$. For a detailed treatment of Knowledge Theory as applied to distributed systems, the reader should refer to the seminal work done in [MDH86,HM90].

Recall that in Section 3.1.2 we defined a failure detector to be a function that maps each failure pattern to a set of failure detector histories. Thus, the specification of a failure detector depends solely on the failure pattern actually encountered. In contrast, the definition of repentance depends on the knowledge (about mistakes) at each process. This in turn depends on the algorithm being executed, and the communication pattern actually encountered. Thus, repentant failure detectors cannot be specified solely in terms of the failure pattern actually encountered. Nevertheless, repentance is an important property that we would like many failure detectors to satisfy.

In the rest of this Appendix, we informally define a hierarchy of repentant failure detectors that differ by their accuracy (i.e., the maximum number of mistakes they can make). As we just noted, such failure detectors cannot be specified solely in terms of the failure pattern actually encountered, and thus they do not fit the formal definition of failure detectors given in Section 3.1.2.

A.2 A hierarchy of repetant failure detectors

We now define an infinite hierarchy of repentant failure detectors. Every failure detector in this hierarchy satisfies weak completeness, repentance, and one of the four types of accuracy that we defined in the previous section. We name these failure detectors after the accuracy property that they satisfy:

- $\mathcal{SF}(k)$ denotes a *Strongly k-Mistaken failure detector*,

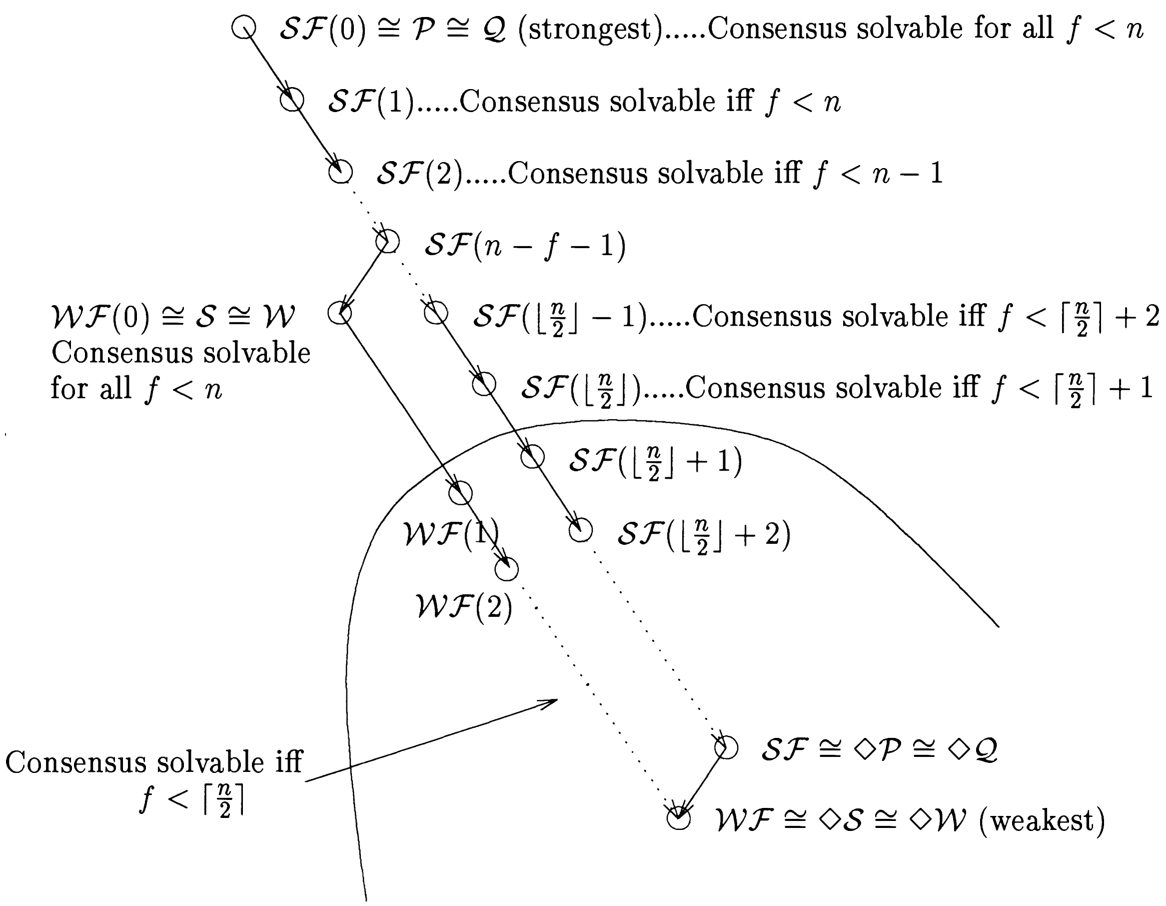

Figure A.1: The hierarchy of repentant failure detectors ordered by reducibility. This figure also shows the maximum number of faulty processes for which Consensus can be solved using each failure detector in this hierarchy.

- $\mathcal{SF}$ denotes a *Strongly Finitely Mistaken failure detector*,

- $\mathcal{WF}(k)$ denotes a *Weakly k-Mistaken failure detector*, and

- $\mathcal{WF}$ denotes a *Weakly Finitely Mistaken failure detector.*

Clearly, $\mathcal{SF}(0) \succeq \mathcal{SF}(1) \succeq \ldots \mathcal{SF}(k) \succeq \mathcal{SF}(k+1) \succeq \ldots \succeq \mathcal{SF}$. A similar order holds for the $\mathcal{WF}$s. Consider a system of n processes of which at most f may crash. In this system, there are at least $n-f$ correct processes. Since $\mathcal{SF}((n-f)-1)$ makes fewer mistakes than the number of correct processes, there is at least one correct process that it never suspects. Thus, $\mathcal{SF}((n-f)-1)$ is weakly 0-mistaken, and $\mathcal{SF}((n-f)-1) \succeq \mathcal{WF}(0)$. Furthermore, it is clear that $\mathcal{SF} \succeq \mathcal{WF}$. This infinite hierarchy of failure detectors, ordered by reducibility, is illustrated in Figure A.1 (where an edge $\rightarrow$ denotes the $\succeq$ relation).

Each of the eight failure detectors that we considered in Section 3.1.5 is equivalent to some failure detector in this hierarchy. In particular, it is easy to show that:

Observation 67:

- $\mathcal{P} \cong \mathcal{Q} \cong \mathcal{SF}(0)$,

- $\mathcal{S} \cong \mathcal{W} \cong \mathcal{WF}(0)$,

- $\Diamond\mathcal{P} \cong \Diamond\mathcal{Q} \cong \mathcal{SF}$, and

- $\Diamond\mathcal{S} \cong \Diamond\mathcal{W} \cong \mathcal{WF}$.

For example, it is easy to see that the reduction algorithm in Figure 3.3 transforms $\mathcal{WF}$ into $\Diamond\mathcal{W}$. Other conversions are similar or straightforward and are therefore omitted. Note that $\mathcal{P}$ and $\Diamond\mathcal{W}$ are the strongest and weakest failure detectors in this hierarchy, respectively. From Corollaries 15 and 31, and Observation 67 we have:

Corollary 68: Given $\mathcal{WF}(0)$, Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < n$.

Similarly, from Corollaries 20 and 32, and Observation 67 we have:

Corollary 69: Given $\mathcal{WF}$, Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < \frac{n}{2}$.

A.3 Tight bounds on fault-tolerance

Since Consensus and Atomic Broadcast are equivalent in asynchronous systems with any number of faulty processes (Theorem 30), we can focus on establishing fault-tolerance bounds for Consensus. In Section 3.5, we showed that failure detectors with *perpetual* accuracy (i.e., $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, or $\mathcal{W}$) can be used to solve Consensus in asynchronous systems with *any* number of failures. In contrast, with failure detectors with *eventual* accuracy (i.e., $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, or $\diamond\mathcal{W}$), Consensus can be solved if and only if a majority of the processes are correct. We now refine this result by considering each failure detector $\mathcal{D}$ in our infinite hierarchy of failure detectors, and determining how many correct processes are necessary to solve Consensus using $\mathcal{D}$. The results are illustrated in Figure A.1.

There are two cases depending on whether we assume that the system has a majority of correct processes or not. Since $\diamond\mathcal{W}$, the weakest failure detector in the hierarchy, can be used to solve Consensus when a majority of the processes are correct, we have:

Observation 70: If $f < \frac{n}{2}$ then Consensus can be solved using any failure detector in the hierarchy of Figure A.1.

We now consider the solvability of Consensus in systems that do not have a majority of correct processes. For these systems, we determine the maximum m for which Consensus is solvable using $\mathcal{SF}(m)$ or $\mathcal{WF}(m)$. We first show that

Consensus is solvable using $\mathcal{SF}(m)$ if and only if m, the number of mistakes, is less than or equal to $n-f$, the number of correct processes. We then show that Consensus is solvable using $\mathcal{WF}(m)$ if and only if $m=0$.

Theorem 71: Suppose $f \geq \frac{n}{2}$. If $m > n-f$ then there is a Strongly m-Mistaken failure detector $\mathcal{SF}(m)$ such that there is no algorithm A which solves Consensus using $\mathcal{SF}(m)$ in asynchronous systems.

PROOF: [*sketch*] We describe the behaviour of a Strongly m-Mistaken failure detector $\mathcal{SF}(m)$ such that with every algorithm A, there is a run R_A of A using $\mathcal{SF}(m)$ that does not satisfy the specification of Consensus. Since $1 \leq n-f \leq \frac{n}{2}$, we can partition the processes into three sets Π_0, Π_1 and $\Pi_{crashed}$, such that Π_0 and Π_1 are non-empty sets containing $n-f$ processes each, and $\Pi_{crashed}$ is a (possibly empty) set containing the remaining $n-2(n-f)$ processes. For the rest of this proof, we will only consider runs in which all processes in $\Pi_{crashed}$ crash in the beginning of the run. Let $q_0 \in \Pi_0$ and $q_1 \in \Pi_1$. Consider any Consensus algorithm A, and the following two runs of A using $\mathcal{SF}(m)$:

- Run $R_0 = \langle F_0, H_0, I, S_0, T_0 \rangle$: All processes in Π_0 propose 0, and all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1. All processes in Π_0 are correct in F_0, while all the f processes in $\Pi_1 \cup \Pi_{crashed}$ crash in F_0 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_0(t) = \Pi_1 \cup \Pi_{crashed}$. Process q_0 permanently suspects every process in $\Pi_1 \cup \Pi_{crashed}$, i.e., $\forall t \in \mathcal{T} : H_0(q_0, t) = \Pi_1 \cup \Pi_{crashed} = F_0(t)$. No other process suspects any process, i.e., $\forall t \in \mathcal{T}, \forall q \neq q_0 : H_0(q,t) = \emptyset$. In this run, it is clear that $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector.

- Run $R_1 = \langle F_1, H_1, I, S_1, T_1 \rangle$: As in R_0, all processes in Π_0 propose 0, and all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1. All processes in Π_1 are correct in F_1, while all the f processes in $\Pi_0 \cup \Pi_{crashed}$ crash in F_1 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_1(t) = \Pi_0 \cup \Pi_{crashed}$. Process q_1 permanently suspects every process in $\Pi_0 \cup \Pi_{crashed}$, and no other process suspects any process. Clearly, $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector in this run.

Assume, without loss of generality, that both R_0 and R_1 satisfy the specification of Consensus. Let t_0 be the time at which q_0 decides in R_0, and let t_1 be the time at which q_1 decides in R_1. There are three possible cases—in each case we construct a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of algorithm A using $\mathcal{SF}(m)$ such that $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector, but R_A violates the specification of Consensus.

1. In R_0, q_0 decides 1. Let $R_A = \langle F_0, H_0, I_A, S_0, T_0 \rangle$ be a run identical to R_0 except that all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 0. Since in F_0 the processes in $\Pi_1 \cup \Pi_{crashed}$ crash right from the beginning of the run, R_0 and R_A are indistinguishable to q_0. Thus, q_0 decides 1 in R_A (as it did in R_0), thereby violating the uniform validity condition of Consensus.

2. In R_1, q_1 decides 0. This case is symmetric to Case 1.

3. In R_0, q_0 decides 0, and in R_1, q_1 decides 1. Construct $R_A = \langle F_A, H_A, I, S_A, T_A \rangle$ as follows. As before, all processes in Π_0 propose 0, all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1, and all processes in $\Pi_{crashed}$ crash in F_A at the beginning of the run. All messages from processes in Π_0 to those in Π_1 and vice-versa, are delayed until time $t_0 + t_1$. Until time t_0, R_A is identical to R_0, except that the processes in Π_1 do not crash, they are only "very slow" and do not take any steps before time t_0. Thus, until time t_0, q_0 cannot distinguish between R_0 and R_A, and it decides 0 at time t_0 in R_A (as it did in R_0). Note that by time t_0, the failure detector $\mathcal{SF}(m)$ made $n - f$ mistakes in R_A: q_0 erroneously suspected that all processes in Π_1 crashed (while they were only slow).

   From time t_0, the construction of R_A continues as follows.

(a) At time t_0, all processes in Π_0, except q_0, crash in F_A.

(b) From time t_0 to time $t_0 + t_1$, q_1 suspects all processes in $\Pi_0 \cup \Pi_{crashed}$, i.e., $\forall t, t_0 \leq t \leq t_0 + t_1 : H_A(q_1, t) = \Pi_0 \cup \Pi_{crashed}$, and no other process suspects any process. By suspecting all the processes in Π_0, including q_0, the failure detector makes one mistake on process q_1 (about q_0). Thus, by time $t_0 + t_1$, $\mathcal{SF}(m)$ has made a total of $(n - f) + 1$ mistakes in R_A. Since $m > n - f$, $\mathcal{SF}(m)$ has made at most m mistakes in R_A until time $t_0 + t_1$.

(c) At time t_0, processes in Π_1 "wake up." From time t_0 to time $t_0 + t_1$ they execute exactly as they did in R_1 from time 0 to time t_1 (they cannot perceive this real-time shift of t_0). Thus, at time $t_0 + t_1$ in run R_A, q_1 decides 1 (as it did at time t_1 in R_1). So q_0 and q_1 decide differently in R_A, and this violates the agreement condition of Consensus.

(d) From time $t_0 + t_1$ onwards the run R_A continues as follows. No more processes crash and every correct process suspects exactly all the processes that have crashed. Thus, $\mathcal{SF}(m)$ satisfies weak completeness, repentance, and makes no further mistakes.

By (b) and (d), $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector in run R_A. From (c), R_A, a run of A that uses $\mathcal{SF}(m)$, violates the specification of Consensus. □

We now show that the above lower bound is tight: Given $\mathcal{SF}(m)$, Consensus can be solved in asynchronous systems with $m \leq n - f$.

Theorem 72: If $m \leq n - f$ then Consensus can be solved in asynchronous systems using any Strongly m-Mistaken failure detector $\mathcal{SF}(m)$.

PROOF: Suppose $m < n - f$. Since m, the number of mistakes made by $\mathcal{SF}(m)$, is less than the number of correct processes, there is at least one correct process that $\mathcal{SF}(m)$ never suspects. Thus, $\mathcal{SF}(m)$ satisfies weak accuracy. By definition, $\mathcal{SF}(m)$ also satisfies weak completeness. So $\mathcal{SF}(m)$ is a Weak Failure Detector and can be used to solve Consensus (Corollary 15).

Suppose $m = n - f$. Even though $\mathcal{SF}(m)$ can now make a mistake on *every* correct process, it can still be used to solve Consensus (even if a majority of the processes are faulty). The algorithm uses rotating coordinators, and is similar to the one for $\diamond\mathcal{W}$ in Figure 3.6. Because of this similarity, we omit the details from this Appendix. $\square$

From the above two theorems:

Corollary 73: Suppose $f \geq \frac{n}{2}$. Consensus can be solved in asynchronous systems using any $\mathcal{SF}(m)$ if and only if $m \leq n - f$.

We now turn our attention to Weakly k-Mistaken failure detectors.

Theorem 74: Suppose $f \geq \frac{n}{2}$. If $m > 0$ then there is a Weakly m-Mistaken failure detector $\mathcal{WF}(m)$ such that there is no algorithm A which solves Consensus using $\mathcal{WF}(m)$ in asynchronous systems.

PROOF: In Theorem 71, we described a failure detector that cannot be used to solve Consensus in asynchronous systems with $f \geq \frac{n}{2}$. It is easy to verify that this failure detector makes at most one mistake about each correct process, and thus it is a Weakly m-Mistaken failure detector. $\square$

From Corollary 68 and the above theorem, we have:

Corollary 75: Suppose $f \geq \frac{n}{2}$. Consensus can be solved in asynchronous systems using any $\mathcal{WF}(m)$ if and only if $m = 0$.

Bibliography

[ABD+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, October 1987.

[ADKM91] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. Technical Report CS91-13, Computer Science Department, The Hebrew University of Jerusalem, November 1991.

[ADLS91] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainity. In *Proceedings of the Twenty third ACM Symposium on Theory of Computing*, May 1991.

[BCJ+90] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Kenneth P. Kane, and Frank Bernhard Schmuck. *ISIS - A Distributed Programming Environment*, June 1990.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus. In *Proceedings of the Thirtieth Symposium on Foundations of Computer Science*, pages 410–415. IEEE Computer Society Press, October 1989.

[BGT90] Navin Budhiraja, Ajei Gopal, and Sam Toueg. Early-stopping distributed bidding and applications. In *Proceedings of the Fourth International Workshop on Distributed Algorithms.* Springer-Verlag, September 1990. In press.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987.

[CASD85] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CDD90] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. Technical Report RJ 7424, IBM Research Laboratory, April 1990.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cri87] Flaviu Cristian. Issues in the design of highly available computing services. In *Annual Symposium of the Canadian Information Processing Society*, pages 9–16, July 1987. Also IBM Research Report RJ5856, July 1987.

[CT90] Tushar Deepak Chandra and Sam Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the Fourth International Workshop on Distributed Algorithms*. Springer-Verlag, September 1990. In press.

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

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

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

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Department of Computer Science, Yale University, June 1983.

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

[GSTC90] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 297–310, August 1990.

[Had84] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* Ph.D. dissertation, Harvard University, June 1984. Department of Computer Science Technical Report 11-84.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[LA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[LF82] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, April 1982.

[Lo93] Wai Kau Lo. Using failure detectors to solve consensus in asynchronous shared-memory systems. Masters dissertation, University of Toronto, January 1993.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MDH86] Yoram Moses, Danny Dolev, and Joseph Y. Halpern. Cheating husbands and other stories: a case study of knowledge, action, and communication. *Distributed Computing*, 1(3):167–176, 1986.

[MSF87] C. Mohan, R. Strong, and S. Finkelstein. Methods for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987.

[Mul87] Sape J. Mullender, editor. *The Amoeba distributed operating system: Selected papers 1984 - 1987.* Centre for Mathematics and Computer Science, 1987.

[PBS89] L. L. Peterson, N. C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. In *ACM Transactions on computer systems 7,3*, pages 217–246, August 1989.

[PGM89] Frank Pittelli and Hector Garcia-Molina. Reliable scheduling in a tmr database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[Pow91] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.

[PSL80] M. Pease, R. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

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

[RB91] Aleta Ricciardi and Ken Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351. ACM Press, August 1991.

[Rei82] Rüdiger Reischuk. A new solution for the Byzantine general's problem. Technical Report RJ 3673, IBM Research Laboratory, November 1982.

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

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.