Unreliable Failure Detectors for Reliable Distributed Systems

Tushar Deepak Chandra
I.B.M Thomas J. Watson Research Center, Hawthorne, New York
and
Sam Toueg
Cornell University, Ithaca, New York

We introduce the concept of unreliable failure detectors and study how they can be used to solve Consensus in asynchronous systems with crash failures. We characterise unreliable failure detectors in terms of two properties — completeness and accuracy. We show that Consensus can be solved even with unreliable failure detectors that make an infinite number of mistakes, and determine which ones can be used to solve Consensus despite any number of crashes, and which ones require a majority of correct processes. We prove that Consensus and Atomic Broadcast are reducible to each other in asynchronous systems with crash failures; thus the above results also apply to Atomic Broadcast. A companion paper shows that one of the failure detectors introduced here is the weakest failure detector for solving Consensus [Chandra et al. 1992].

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: *reliability, availability, and serviceability*; D.1.3 [**Programming Techniques**]: Concurrent programming—*distributed programming*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata; relations among models*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; H.2.4 [**Database Management**]: Systems—*concurrency; distributed systems; transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: agreement problem, asynchronous systems, atomic broadcast, *Byzantine Generals'* problem, commit problem, consensus problem, crash failures, failure detection, fault-tolerance, message passing, partial synchrony, processor failures

A preliminary version of this paper appeared in *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM press, August 1991.

Research supported by an IBM graduate fellowship, NSF grants CCR-8901780, CCR-9102231, and CCR-940286, and DARPA/NASA Ames Grant NAG-2-593.

Authors' present addresses: Tushar Deepak Chandra, I.B.M T.J. Watson Research Center, 30 Saw Mill Road, Hawthorne, NY 10532; Sam Toueg, Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853.

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 [Wensley et al. 1978], State Machines [Lamport 1978; Schneider 1990], Isis [Birman and Joseph 1987; Birman et al. 1990], Psync [Peterson et al. 1989], Amoeba [Mullender 1987], Delta-4 [Powell 1991], Transis [Amir et al. 1991], HAS [Cristian 1987], FAA [Cristian et al. 1990], 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 paper, 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 [Fischer et al. 1985; Dolev et al. 1987].[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 randomisation techniques [Chor and Dwork 1989], the definition of some weaker problems and their solutions [Dolev et al. 1986; Attiya et al. 1987; Bridgland and Watro 1987; Biran et al. 1988], or the study of several models of *partial synchrony* [Dolev et al. 1987; Dwork et al. 1988]. 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 paper, 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

[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.

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. In the rest of this introduction, we informally describe this concept and summarise our results.

We consider *distributed* failure detectors: each process has access to a local *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 paper. 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 the underlying failure detection mechanisms, such as the *Isis Toolkit* [Birman et al. 1990] for asynchronous fault-tolerant distributed computing.

We characterise a class of failure detectors by specifying the *completeness* and *accuracy* properties that failure detectors in this class must satisfy. Roughly speaking, *completeness* requires that a failure detector eventually suspects every process that actually crashes,[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 classes of failure detectors, and consider the problem of solving Consensus using failure detectors from each class.[4]

[2] A different approach was taken by the Isis system [Ricciardi and Birman 1991]: a correct process that is wrongly suspected to have crashed, is forced to crash itself. 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 9.3.

[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:

To do so, we 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 that can transform $\mathcal{D}$ into $\mathcal{D}'$. We also say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$: 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 the concept of reducibility (extended to classes of failure detectors), we show how to reduce our eight classes of failure detectors to four, and consider how to solve Consensus for each class.

We show that certain failure detectors can be used to solve Consensus in systems with any number of process failures, while others require a majority of correct processes. In order to better understand where the majority requirement becomes necessary, we study an infinite hierarchy of failure detector classes and determine exactly where in this hierarchy the majority requirement becomes necessary.

Of special interest is $\diamond\mathcal{W}$, the weakest class of failure detectors considered in this paper. Informally, a failure detector is in $\diamond\mathcal{W}$ if it satisfies the following two properties:

Completeness. There is a time after which every process that crashes is permanently suspected by some correct process.

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

Such a failure detector can make an infinite number of mistakes: Each local failure detector module can repeatedly add and then remove *correct* processes from its list of suspects (this reflects the inherent difficulty of determining whether a process 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 some conditions must hold forever; of course this cannot be achieved in a real system. However, in practice it is not really required that these conditions hold forever. When solving a problem that "terminates", such as Consensus, it is enough that they hold for a "sufficiently long" period of time: This period should be long enough for the algorithm to achieve its goal (e.g., for correct processes to decide). When solving a problem that does not terminate, such as Atomic Broadcast, it is enough that these properties hold for "sufficiently long" periods of time: Each period should be long enough for some progress to occur (e.g., for correct processes to deliver some messages). However, in an asynchronous system it is not possible to quantify "sufficiently long", since even a single process step 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.[5]

Another desirable feature of $\diamond\mathcal{W}$ is the following. If an application assumes a failure detector with the properties of $\diamond\mathcal{W}$, but the failure detector that it actually

---

any Consensus algorithm can be transformed into an Atomic Broadcast algorithm and vice versa. Thus, we can focus on solving Consensus since all our results will automatically apply to Atomic Broadcast as well.

[5]Solving a problem with the assumption that certain properties hold for sufficiently long has been done previously, see [Dwork et al. 1988].

uses "malfunctions" and continuously fails to meet these properties — for example, there is a crash that no process ever detects, and all correct processes are repeatedly (and forever) falsely suspected — the application may lose *liveness* but not *safety*. For example, if a Consensus algorithm assumes the properties of $\diamond\mathcal{W}$, but the failure detector that it actually uses misbehaves continuously, processes may be prevented from deciding, but they never decide different values (or a value that is not allowed). Similarly, with an Atomic Broadcast algorithm, processes may stop delivering messages, but they never deliver messages out-of-order.

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 based on an idea in [Dwork et al. 1988] (a more detailed description of this implementation and of its properties is given in Section 9.1).

Every process q periodically sends a "q-is-alive" message to all. If a process p times-out on some process q, it adds q to its list of suspects. If p later receives a "q-is-alive" message, p recognises that it made a mistake by prematurely timing out on q: p removes q from its list of suspects, and increases the length of its timeout period for q in an attempt to prevent a similar mistake in the future.

In an asynchronous system, this scheme does not implement a failure detector with the properties of $\diamond\mathcal{W}$:[6] an unbounded sequence of premature time-outs may cause every correct process to be repeatedly added and then removed from the list of suspects of every correct process, thereby violating the accuracy property of $\diamond\mathcal{W}$. Nevertheless, in many practical systems, increasing the timeout period after each mistake ensures 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 for periods that are "long enough" for the application using the failure detector to make sufficient progress or 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" periods of time.

Having made the point that in practical systems one can use time-outs to implement a failure detector with the properties of $\diamond\mathcal{W}$, 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 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.

Any failure detector that satisfies the completeness and accuracy properties of

[6] Indeed, no algorithm can implement such a failure detector in an asynchronous system: as we show in Section 6.2, this implementation could be used to solve Consensus in such a system, contradicting the impossibility result of [Fischer et al. 1985].

$\diamond\mathcal{W}$ provides *sufficient* information about failures to solve Consensus. But is this information *necessary*? Indeed, what it is the "weakest" failure detector for solving Consensus?

[Chandra et al. 1992] answer this question by considering $\diamond\mathcal{W}_0$, the weakest failure detector in $\diamond\mathcal{W}$. Roughly speaking, $\diamond\mathcal{W}_0$ satisfies the properties of $\diamond\mathcal{W}$, *and no other properties*. [Chandra et al. 1992] show that $\diamond\mathcal{W}_0$ is the weakest failure detector that can be used to solve Consensus in asynchronous systems (with a majority of correct processes). More precisely, [Chandra et al. 1992] show that if a failure detector $\mathcal{D}$ can be used to solve Consensus, then there is a distributed algorithm that transforms $\mathcal{D}$ into $\diamond\mathcal{W}_0$. Thus, in a precise sense, $\diamond\mathcal{W}_0$ 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-tolerant distributed computing in asynchronous systems.

In our discussion so far, we focused on the Consensus problem. In Section 7, 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.[7] In other words, a solution for one automatically yields a solution for the other. Thus, Atomic Broadcast can be solved using the unreliable failure detectors described in this paper. Furthermore, $\diamond\mathcal{W}_0$ 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 [Dolev et al. 1987] and [Dwork et al. 1988]. 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 at least 34 different models of partial synchrony and for each model determine whether or not Consensus can be solved. In this paper, we argue that partial synchrony assumptions can be encapsulated in the unreliability of failure detectors. For example, in the models of partial synchrony considered in [Dwork et al. 1988] it is easy to implement a failure detector that satisfies the properties of $\diamond\mathcal{W}$. 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.[8]

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.

Finally, even though this paper is concerned with solvability rather than effi-

[7] 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 paper.

[8] The relation between our approach and partial synchrony is discussed in more detail in Section 9.1.

ciency, one of our algorithms (the one assuming a failure detector with the properties of $\diamond\mathcal{W}$) appears to be quite efficient: We have recently implemented a slightly modified version that achieves Consensus within two "asynchronous rounds" in most runs. Thus, we believe that unreliable failure detectors can be used to bridge the gap between known impossibility results and the need for practical solutions for fault-tolerant asynchronous systems.

The remainder of this paper is organised as follows. In Section 2, we describe our model and introduce eight classes of failure detectors defined in terms of properties. In Section 3, we use the concept of reduction to show that we can focus on four classes of failure detectors rather than eight. In Section 4, we present *Reliable Broadcast*, a communication primitive for asynchronous systems used by several of our algorithms. In Section 5, we define the Consensus problem. In Section 6, we show how to solve Consensus for each one of the four equivalence classes of failure detectors. In Section 7, we show that Consensus and Atomic Broadcast are equivalent to each other in asynchronous systems. In Section 8, we complete our comparison of the failure detector classes defined in this paper. In Section 9, we discuss related work, and in particular, we describe an implementation of a failure detector with the properties of $\diamond\mathcal{W}$ in several models of partial synchrony. Finally, in the Appendix we define an infinite hierarchy of failure detector classes, and determine exactly where in this hierarchy a majority of correct processes is required to solve Consensus.

2. 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 [Fischer et al. 1985]. 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.

2.1 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., $\mathit{correct}(F) \neq \emptyset$.

2.2 Failure detectors

Each failure detector module outputs the set of processes that it currently suspects to have crashed.[9] 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

[9] In [Chandra et al. 1992] failure detectors can output values from an *arbitrary* range.

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 $\mathcal{D}$.[10]

In this paper, 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 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.

2.3 Failure detector properties

We now state two completeness properties and four accuracy properties that a failure detector $\mathcal{D}$ may satisfy.

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.

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:

[10] 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 have a different failure detector history.

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.

2.4 Failure detector classes

A failure detector is said to be *Perfect* if it satisfies strong completeness and strong accuracy. The set of all such failure detectors, called the *class of Perfect failure detectors*, is denoted by $\mathcal{P}$. Similar definitions arise for each pair of completeness and accuracy properties. There are eight such pairs, obtained by selecting one of the two completeness properties and one of the four accuracy properties introduced in the previous section. The resulting definitions and corresponding notation are given in Figure 1.

2.5 Algorithms and runs

In this paper, we focus on algorithms that use unreliable failure detectors. To describe such algorithms, we only need informal definitions of algorithms and runs,

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

Fig. 1. Eight classes of failure detectors defined in terms of accuracy and completeness.

based on the formal definitions given in [Chandra et al. 1992].[11]

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 to a single process.[12] Since we model asynchronous systems, messages may experience arbitrary (but finite) delays. Furthermore, there is no bound on relative process speeds.

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.

Informally, a *problem* P is defined by a set of properties that runs must satisfy. An algorithm A *solves a problem P using a failure detector $\mathcal{D}$* if all the runs of A using $\mathcal{D}$ satisfy the properties required by P. Let $\mathcal{C}$ be a class of failure detectors. Algorithm A *solves problem P using* $\mathcal{C}$ if for all $\mathcal{D} \in \mathcal{C}$, A solves P using $\mathcal{D}$. Finally, we say that problem P *can be solved using* $\mathcal{C}$ if for all failure detectors $\mathcal{D} \in \mathcal{C}$, there is an algorithm A that solves P using $\mathcal{D}$.

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 $H_{\mathcal{D}}(p, t)$.

2.6 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 $\mathcal{D}'$ ($T_{\mathcal{D}\to\mathcal{D}'}$ is called a *reduction algorithm*). Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ to maintain a variable $output_p$ at every process p. This variable, which is part of the local state of p, emulates the output of $\mathcal{D}'$ at p. Algo-

[11] Formal definitions are necessary in [Chandra et al. 1992] to prove a subtle lower bound.

[12] [Chandra et al. 1992] assume that each step is *atomic*, i.e., indivisible with respect to failures. Furthermore, each process can send a message to *all* processes during such a step. These assumptions were made to strengthen the lower bound result of [Chandra et al. 1992].

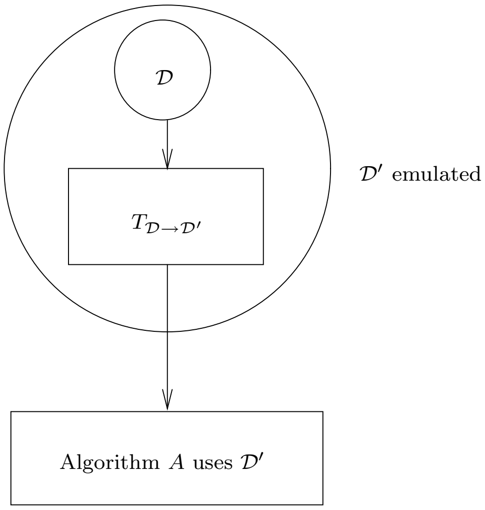

Fig. 2. Transforming $\mathcal{D}$ into $\mathcal{D}'$.

rithm $T_{\mathcal{D}\to\mathcal{D}'}$ *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 $\mathcal{D}$, $output^R \in \mathcal{D}'(F)$. Note that $T_{\mathcal{D}\to\mathcal{D}'}$ 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 $\mathcal{D}'$.

Given a reduction algorithm $T_{\mathcal{D}\to\mathcal{D}'}$, any problem that can be solved using failure detector $\mathcal{D}'$, can be solved using $\mathcal{D}$ instead. To see this, suppose a given algorithm A requires failure detector $\mathcal{D}'$, but only $\mathcal{D}$ is available. We can still execute A as follows. Concurrently with A, processes run $T_{\mathcal{D}\to\mathcal{D}'}$ to transform $\mathcal{D}$ into $\mathcal{D}'$. We modify algorithm A at process p as follows: whenever A requires that p queries 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 2.

Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ must provide 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* $\mathcal{D}$. Clearly, $\succeq$ is a transitive relation. 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*.

Similarly, given two classes of failure detectors $\mathcal{C}$ and $\mathcal{C}'$, if for each failure detector $\mathcal{D} \in \mathcal{C}$ there is a failure detector $\mathcal{D}' \in \mathcal{C}'$ such that $\mathcal{D} \succeq \mathcal{D}'$, we write $\mathcal{C} \succeq \mathcal{C}'$ and say that $\mathcal{C}'$ *is weaker than* $\mathcal{C}$ (note that if $C \succeq C'$, then if a problem is solvable using C', it is also solvable using C). From this definition, $\succeq$ is clearly transitive. If $\mathcal{C} \succeq \mathcal{C}'$ and $\mathcal{C}' \succeq \mathcal{C}$, we write $\mathcal{C} \cong \mathcal{C}'$ and say that $\mathcal{C}$ and $\mathcal{C}'$ are *equivalent*.

Consider the trivial reduction algorithm in which each process p periodically writes the current value output by its local failure detector module into $output_p$. From this trivial reduction the following relations between classes of failure detectors are immediate:

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

Every process p executes the following:

$output_p \leftarrow \emptyset$

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

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

Fig. 3. $T_{\mathcal{D}\rightarrow\mathcal{D}'}$: From Weak Completeness to Strong Completeness.

3. FROM WEAK COMPLETENESS TO STRONG COMPLETENESS

In Figure 3, we give a reduction algorithm $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ that transforms any given failure detector $\mathcal{D}$ that satisfies weak completeness, into a failure detector $\mathcal{D}'$ that satisfies strong completeness. Furthermore, if $\mathcal{D}$ satisfies one of the four accuracy properties that we defined in Section 2.3 then $\mathcal{D}'$ also does so. In other words, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ strengthens completeness while preserving accuracy.

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

Informally, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ 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 $\mathcal{D}_p$ — to every process. When p receives a message of the form $(q, suspects_q)$, it adds $suspects_q$ to $output_p$ and removes q from $output_p$ (recall that $output_p$ is the variable emulating the output of the failure detector module $\mathcal{D}'_p$).

In our algorithms, we use the notation "send m to all" as a short-hand for "for all $q \in \Pi$: send m to q." If a process p crashes while executing this "for loop", it is possible that some processes receive the message m while others do not.

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:

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

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:

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

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:

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

Lemma 1. *$T_{\mathcal{D}\to\mathcal{D}'}$ 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 $(q, \mathit{suspects}_q)$ with $p \in \mathit{suspects}_q$ to all processes. Eventually, every correct process receives $(q, \mathit{suspects}_q)$ and adds p to output (in 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 2. *$T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P2.*

Proof. Let p be any process. Suppose 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. □

Lemma 3. *$T_{\mathcal{D}\to\mathcal{D}'}$ 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. □

Theorem 1. *$\mathcal{Q} \succeq \mathcal{P}$, $\mathcal{W} \succeq \mathcal{S}$, $\Diamond\mathcal{Q} \succeq \Diamond\mathcal{P}$, and $\Diamond\mathcal{W} \succeq \Diamond\mathcal{S}$.*

Proof. Let $\mathcal{D}$ be any failure detector in $\mathcal{Q}$, $\mathcal{W}$, $\Diamond\mathcal{Q}$, or $\Diamond\mathcal{W}$. We show that $T_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{D}$ into a failure detector $\mathcal{D}'$ in $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{P}$, or $\Diamond\mathcal{S}$, respectively. Since $\mathcal{D}$ satisfies weak completeness, by Lemma 1, $\mathcal{D}'$ satisfies strong completeness.

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
    $R\text{-}deliver(m)$

Fig. 4. Reliable Broadcast by message diffusion.

We now show that $\mathcal{D}$ and $\mathcal{D}'$ have the same accuracy property. If $\mathcal{D}$ is in $\mathcal{Q}$ or $\mathcal{W}$, this follows from Lemma 2. If $\mathcal{D}$ is in $\diamond\mathcal{Q}$ or $\diamond\mathcal{W}$, this follows from Lemma 3. □

By Theorem 1 and Observation 1, we have:

Corollary 1. *$\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$, and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$.*

The relations given in Corollary 1 are sufficient for the purposes of this paper. A complete enumeration of the relations between the eight failure detectors classes defined in Figure 1 is given in Section 8.

4. Reliable Broadcast

We now define Reliable Broadcast, a communication primitive for asynchronous systems that we use in our algorithms.[13] 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. We assume that every message m includes a field denoted $sender(m)$ that contains the identity of the sender, and a field with a sequence number; these two fields make every message unique. Reliable Broadcast satisfies the following properties [Hadzilacos and Toueg 1994]:

Validity. If a correct process R-broadcasts a message m, then it eventually R-delivers 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, every process R-delivers m at most once, and only if m was previously R-broadcast by $sender(m)$.

In Figure 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. This algorithm satisfies validity,

[13]This is a crash-failure version of the asynchronous broadcast primitive defined in [Bracha and Toueg 1985] for “Byzantine” failures.

agreement and uniform integrity in asynchronous systems with up to $n-1$ crash failures. The proof is obvious and therefore omitted.

5. 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 [Fischer 1983]. 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 integrity. Every process decides at most once.

Agreement. No two correct processes decide differently.

Uniform validity. If a process decides v, then v was proposed by some process.[14]

It is well-known that Consensus cannot be solved in asynchronous systems that are subject to even a single crash failure [Fischer et al. 1985; Dolev et al. 1987].

6. SOLVING CONSENSUS USING UNRELIABLE FAILURE DETECTORS

We now show how to solve Consensus using each one of the eight classes of failure detectors defined in Figure 1. By Corollary 1, we only need to show how to solve Consensus using each one of the four classes of failure detectors that satisfy strong completeness, namely, $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$.

In Section 6.1, we present an algorithm that solves Consensus using $\mathcal{S}$. Since $\mathcal{P} \succeq \mathcal{S}$, this algorithm also solves Consensus using $\mathcal{P}$. In Section 6.2, we give a Consensus algorithm that uses $\diamond\mathcal{S}$. Since $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$, this algorithm also solves Consensus using $\diamond\mathcal{P}$. Our Consensus algorithms actually solve a stronger form of Consensus than the one specified in Section 5: They ensure that no two processes, *whether correct or faulty*, decide differently — a property called *uniform agreement* [Neiger and Toueg 1990].

The Consensus algorithm that uses $\mathcal{S}$ tolerates any number of failures. In contrast, the one that uses $\diamond\mathcal{S}$ requires a majority of correct processes. We show that to solve Consensus this requirement is necessary even if one uses $\diamond\mathcal{P}$, a class of failure detectors 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.

6.1 Solving Consensus using $\mathcal{S}$

The algorithm in Figure 5 solves Consensus using any Strong failure detector $\mathcal{D} \in \mathcal{S}$. In other words, it works with any failure detector $\mathcal{D}$ that satisfies strong completeness and weak accuracy. This algorithm tolerates up to $n-1$ faulty processes (in asynchronous systems with n processes).

[14] The validity property captures the relation between the decision value and the proposed values. Changing this property results in other types of Consensus [Fischer 1983].

Every process p executes the following:

procedure *propose*(v_p)
  $V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$ {*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** $q \in \mathcal{D}_p$] {*query the failure detector*}
    $msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$
    $\Delta_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$
    for $k \leftarrow 1$ to n
      if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ with $\Delta_q[k] \neq \perp$ **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** $q \in \mathcal{D}_p$] {*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 $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

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

Fig. 5. Solving Consensus using any $\mathcal{D} \in \mathcal{S}$.

The 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 $\mathcal{D}_p$, before proceeding to round $r+1$. Note that while p is waiting for a message from q in round r, it is possible that q is added to $\mathcal{D}_p$. If this occurs, 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 $R = \langle F, H_{\mathcal{D}}, I, S, T \rangle$ be any run of the algorithm in Figure 5 using $\mathcal{D} \in \mathcal{S}$ in which all correct processes propose a value. We have to show that the termination, uniform validity, agreement and uniform integrity properties of Consensus hold.

Note that $V_p[q]$ is p's current estimate of q's proposed value. Furthermore, $\Delta_p[q] = v_q$ 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.

Lemma 4. *For all p and q, and in all phases, $V_p[q]$ is either v_q or $\perp$.*

Proof. Obvious from the algorithm. □

Lemma 5. *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 $\mathcal{D}_p$. There are two cases to consider:

(1) q crashes. Since $\mathcal{D}$ satisfies strong completeness, there is a time after which $q \in \mathcal{D}_p$.

(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. □

Since $\mathcal{D}$ satisfies weak accuracy there is a correct process c that is never suspected by any process, i.e., $\forall t \in \mathcal{T}, \forall p \in \Pi - F(t) : c \notin H_{\mathcal{D}}(p,t)$. 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 6. *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 $c \notin \mathcal{D}_p$, p waits for and receives the message (r, Δ_c, c) from c. □

Lemma 7. *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 4, $V_c[q] = v_q$. 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 6, 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 8. *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 7, 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) $V_c[q] = \bot$ at the end of Phase 1. Since $c \notin \mathcal{D}_p$, p waits for and receives V_c in Phase 2. Since $V_c[q] = \bot$, p sets $V_p[q] \leftarrow \bot$ at the end of Phase 2. □

LEMMA 9. *(Uniform agreement) No two processes decide differently.*

PROOF. From Lemma 8, all processes that reach Phase 3 have the same vector V. Thus, all processes that decide, decide the same value. □

LEMMA 10. *For all $p \in \Pi_2$, $V_p[c] = v_c$ at the end of Phase 2.*

PROOF. From the algorithm, $V_c[c] = v_c$ at the end of Phase 1. From Lemma 7, for all $q \in \Pi_1$, $V_q[c] = v_c$ at the end of Phase 1. Thus, no process sends V with $V[c] = \bot$ 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 2. *The algorithm in Figure 5 solves Consensus using $\mathcal{S}$ in asynchronous systems.*

PROOF. From the algorithm in Figure 5, it is clear that no process decides more than once, and this satisfies the uniform integrity requirement of Consensus. By Lemma 9, the (uniform) agreement property of Consensus holds. From Lemma 5, every correct process eventually reaches Phase 3. From Lemma 10, the vector V_p of every correct process has at least one non-$\bot$ 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-$\bot$ component of V_p. Thus, every correct process decides some non-$\bot$ value in Phase 3—and this satisfies termination of Consensus. From Lemma 4, this non-$\bot$ decision value is the proposed value of some process. Thus, uniform validity of Consensus is also satisfied. □

By Theorems 1 and 2, we have:

COROLLARY 2. *Consensus is solvable using $\mathcal{W}$ in asynchronous systems.*

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

In the previous section, we showed how to solve Consensus using $\mathcal{S}$, a class of failure detectors that satisfy weak accuracy: at least one correct process is *never* suspected. That solution tolerates any number of process failures. If we assume that the maximum number of faulty processes is less than half then we can solve Consensus using $\diamond\mathcal{S}$, a class of failure detectors that satisfy only *eventual* weak accuracy. With such failure detectors, *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 determine whether any specific process is correct, or whether some correct process will never be suspected after time t.)

Let f denote the maximum number of processes that may crash.[15] Consider asynchronous systems with $f < \lceil n/2 \rceil$, i.e., where at least $\lceil (n+1)/2 \rceil$ processes are correct. In such systems, the algorithm in Figure 6 solves Consensus using any Eventual Strong failure detector $\mathcal{D} \in \diamond\mathcal{S}$. In other words, it works with any failure detector $\mathcal{D}$ that satisfies strong completeness and eventual weak accuracy.

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

This algorithm uses the *rotating coordinator* paradigm [Reischuk 1982; Chang and Maxemchuk 1984; Dwork et al. 1988; Berman et al. 1989; Chandra and Toueg 1990], and it 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 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.[16]

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 $\lceil (n+1)/2 \rceil$ 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 $\mathcal{D}_p$, p *suspects* that c crashed, and sends a *nack* to c.

In Phase 4, c waits for $\lceil (n+1)/2 \rceil$ replies (*ack*s or *nack*s). If all replies are *ack*s, then c knows that a majority of 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.

This algorithm relies on the assumption that $f < \lceil n/2 \rceil$, i.e., that at least $\lceil (n+1)/2 \rceil$ processes are correct. Note that processes do not have to know the value of f. But they do need to have *a priori* knowledge of the list of (potential) coordinators. Let R be any run of the algorithm in Figure 6 using $\mathcal{D} \in \Diamond\mathcal{S}$ in which all correct processes propose a value. We have to show that the termination, uniform validity, agreement and uniform integrity properties of Consensus hold.

LEMMA 11. *(Uniform agreement) No two processes decide differently.*

PROOF. If no process ever decides, the lemma is trivially true. If any process decides, it must have previously R-delivered a message of the type $(-,-,-,decide)$. By the uniform integrity property of Reliable Broadcast and the algorithm, a coordinator previously R-broadcast this message. This coordinator must have received $\lceil (n+1)/2 \rceil$ messages of the type $(-,-,ack)$ in Phase 4. Let r be the smallest round number in which $\lceil (n+1)/2 \rceil$ 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

[16]Many Consensus algorithms in the literature have the property that a value gets locked before processes decide, e.g. [Reischuk 1982; Dwork et al. 1988].

Every process p executes the following:

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

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

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

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

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

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

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

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

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

Fig. 6. Solving Consensus using any $\mathcal{D} \in \diamond\mathcal{S}$.

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 $\lceil (n+1)/2 \rceil$ processes. Thus, there is some process p such that (1) p sent a (p, r, ack) message to c in Phase 3 of round r, and (2) $(p, k, estimate_p, ts_p)$ is in $msgs_{c_k}[k]$ in Phase 2 of round k. Since p sent (p, r, ack) to c in Phase 3 of round r, $ts_p = r$ at the end of Phase 3 of round r. Since ts_p is 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 $(q, k, estimate_q, t)$ is in $msgs_{c_k}[k]$. From Figure 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 $estimate_q$. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast $(q, r_q, estimate_q, decide)$ in Phase 4 of round r_q. From Figure 6, q must have received $\lceil (n+1)/2 \rceil$ 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, $estimate_q = estimate_c$. $\square$

LEMMA 12. *(Termination) 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 $c = (r \bmod n) + 1$. Since a majority of the processes are correct, at least $\lceil (n+1)/2 \rceil$ 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 eventually receives $(c, r, estimate_c)$. In the second case, since $\mathcal{D}$ satisfies strong completeness, for every correct process p there is a time after which c is permanently suspected by p, i.e., $c \in \mathcal{D}_p$. 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 at least $\lceil (n+1)/2 \rceil$ 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 $\mathcal{D}$ satisfies eventual weak accuracy, there is a correct process q and a time t such that no correct process suspects q after t. Let $t' \geq t$ be a time such that all faulty processes crash. Note that after time t' *no* process suspects q. 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 $\lceil (n+1)/2 \rceil$ such estimates, and sends $(q, r, estimate_q)$ to all processes. In Phase 3, since q is not suspected by any correct process after time t, every correct process waits for q's estimate, eventually receives it, and replies with an *ack* to q. Furthermore, no process sends a *nack* to q (that can only happen when a process suspects q). Thus in Phase 4, q receives $\lceil (n+1)/2 \rceil$ 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 and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message and *decide*—a contradiction. Thus case 2 is impossible, and this concludes the proof of the lemma. □

THEOREM 3. *The algorithm in Figure 6 solves Consensus using $\diamond\mathcal{S}$ in asynchronous systems with $f < \lceil \frac{n}{2} \rceil$.*

PROOF. Lemma 11 and Lemma 12 show that the algorithm in Figure 6 satisfies the (uniform) agreement and termination properties of Consensus, respectively. From the algorithm, it is clear that no process decides more than once, and hence the uniform integrity property holds. From the algorithm it is also clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, uniform validity of Consensus is also satisfied. □

By Theorems 1 and 3, we have:

COROLLARY 3. *Consensus is solvable using $\diamond\mathcal{W}$ in asynchronous systems with $f < \lceil \frac{n}{2} \rceil$.*

Thus, Consensus can be solved in asynchronous systems using any failure detector in $\diamond\mathcal{W}$, the weakest class of failure detectors considered in this paper. This leads to the following question: What is the weakest failure detector for solving Consensus? The answer to this question, given in a companion paper [Chandra et al. 1992], is summarised below.

Let $\diamond\mathcal{W}_0$ be the "weakest" failure detector in $\diamond\mathcal{W}$. Roughly speaking, $\diamond\mathcal{W}_0$ is the failure detector that exhibits *all* the failure detector behaviours allowed by the properties that define $\Diamond\mathcal{W}$. More precisely, $\Diamond\mathcal{W}_0$ consists of *all* the failure detector histories that satisfy weak completeness and eventual weak accuracy (for a formal definition see [Chandra et al. 1992]). [Chandra et al. 1992] show that $\Diamond\mathcal{W}_0$ is the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes.

THEOREM 4. *[Chandra et al. 1992] If a failure detector $\mathcal{D}$ can be used to solve Consensus in an asynchronous system, then $\mathcal{D} \succeq \Diamond\mathcal{W}_0$ in that system.*

By Corollary 3 and Theorem 4, we have:

COROLLARY 4. *$\Diamond\mathcal{W}_0$ is the weakest failure detector for solving Consensus in asynchronous systems with $f < \lceil \frac{n}{2} \rceil$.*

6.3 A lower bound on fault-tolerance

In Section 6.1, we showed that failure detectors with *perpetual* accuracy (i.e., in $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, 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., in $\Diamond\mathcal{P}$, $\Diamond\mathcal{Q}$, $\Diamond\mathcal{S}$, or $\Diamond\mathcal{W}$), our Consensus algorithms require a majority of the processes to be correct. It turns out that this requirement is necessary: Using $\Diamond\mathcal{P}$ to solve Consensus requires a majority of correct processes. Since $\Diamond\mathcal{P} \succeq \Diamond\mathcal{S}$, the algorithm in Figure 6 is optimal with respect to fault-tolerance.

The proof of this result (Theorem 5) uses standard "partitioning" techniques (e.g., [Ben-Or 1983; Bracha and Toueg 1985]). It is also a corollary of Theorem 4.3 in [Dwork et al. 1988] together with Theorem 9 in Section 9.1.

THEOREM 5. *Consensus cannot be solved using $\Diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$.*

PROOF. We give a failure detector $\mathcal{D} \in \Diamond\mathcal{P}$ such that no algorithm can solve Consensus using $\mathcal{D}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$. Informally $\mathcal{D}$ is the weakest Eventually Perfect failure detector: it consists of *all* failure detector histories that satisfy strong completeness and eventual strong accuracy. More precisely, for every failure pattern F, $\mathcal{D}(F)$ consists of all failure detector histories H such that $\exists t \in \mathcal{T}, \forall t' \geq t, \forall p \in \mathit{correct}(F) : q \in \mathit{crashed}(F) \iff q \in H(p, t')$.

The proof is by contradiction. Suppose algorithm A solves Consensus using $\mathcal{D}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$. 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 the following two runs of A using $\mathcal{D}$:

Run $R_0 = \langle F_0, H_0, I_0, S_0, T_0 \rangle$. All processes propose 0. 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., $\forall t \in \mathcal{T}$, $\forall p \in \Pi_0 : H_0(p, t) = \Pi_1$. Clearly, $H_0 \in \mathcal{D}(F_0)$ as required.

Run $R_1 = \langle F_1, H_1, I_1, S_1, T_1 \rangle$. All processes 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, $H_1 \in \mathcal{D}(F_1)$ as required.

Since R_0 and R_1 are runs of A using $\mathcal{D}$, these runs satisfy the specification of Consensus — in particular, all correct processes decide 0 in R_0, and 1 in R_1. Let $q_0 \in \Pi_0$, $q_1 \in \Pi_1$, t_0 be the time at which q_0 decides 0 in R_0, and t_1 be the time at which q_1 decides 1 in R_1. We now construct a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of algorithm A using $\mathcal{D}$ such that R_A violates the specification of Consensus — a contradiction.

In R_A all processes in Π_0 propose 0 and all processes in Π_1 propose 1. No process crashes in F_A, i.e., $\forall t \in \mathcal{T} : F_A(t) = \emptyset$. 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. More precisely:

$$
\begin{aligned}
&\forall t \leq \max(t_0, t_1) : \\
&\qquad\qquad \forall p \in \Pi_0 : H_A(p, t) = \Pi_1 \\
&\qquad\qquad \forall p \in \Pi_1 : H_A(p, t) = \Pi_0 \\
&\forall t > \max(t_0, t_1), \forall p \in \Pi : H_A(p, t) = \emptyset
\end{aligned}
$$

Note that $H_A \in \mathcal{D}(F_A)$ as required.

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 R_A, q_0 decides 0 at time t_0, while q_1 decides 1 at time t_1. This violates the agreement property of Consensus. □

In the appendix, we refine the result of Theorem 5: We first define an infinite hierarchy of failure detector classes ordered by the maximum number of mistakes that failure detectors can make, and then we show exactly where in this hierarchy the majority requirement becomes necessary for solving Consensus (this hierarchy contains all the eight failure detector classes defined in Figure 1).

7. 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'.

The total order and agreement properties of Atomic Broadcast ensure that all correct processes deliver the same *sequence* of messages. Atomic Broadcast is a powerful communication paradigm for fault-tolerant distributed computing [Chang and Maxemchuk 1984; Cristian et al. 1985; Birman and Joseph 1987; Pittelli and Garcia-Molina 1989; Budhiraja et al. 1990; Gopal et al. 1990; Schneider 1990].

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.[17]

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

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 by a deterministic algorithm in asynchronous systems, even if we assume that at most one process may fail, and it may only fail by crashing. This is because Consensus has no deterministic solution in such systems [Fischer et al. 1985].

(2) Atomic Broadcast can be solved using *randomisation* or *unreliable failure detectors* in asynchronous systems. This is because Consensus is solvable using these techniques in such systems (for a survey of randomised Consensus algorithms, see [Chor and Dwork 1989]).

Consensus can be easily reduced to Atomic Broadcast as follows [Dolev et al. 1987]. 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.[18] 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.

7.1 Reducing Atomic Broadcast to Consensus

In Figure 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.

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

Our Atomic Broadcast algorithm also uses the $\textit{R-broadcast}(m)$ and $\textit{R-deliver}(m)$ 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.

The Atomic Broadcast algorithm described in Figure 7 consists of three tasks, *Task* 1, *Task* 2, and *Task* 3, such that: (1) any task that is enabled is eventually executed, and (2) Task i can execute concurrently with Task j provided $i \neq j$.

When a process wishes to A-broadcast a message m, it R-broadcasts m (Task 1). When a process p R-delivers m, it adds m to the set $\textit{R\_delivered}_p$ (Task 2). When p A-delivers a message m, it adds m to the set $\textit{A\_delivered}_p$ (Task 3). Thus, $\textit{R\_delivered}_p - \textit{A\_delivered}_p$, denoted $\textit{A\_undelivered}_p$, is the set of messages that

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

Every process p executes the following:

Initialisation:

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

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

R-broadcast(m)

A-deliver($-$) 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 $decide(k, msgSet^k)$
$A\_deliver^k \leftarrow msgSet^k - A\_delivered$
atomically deliver all messages in $A\_deliver^k$ in some deterministic order
$A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

Fig. 7. Using Consensus to solve Atomic Broadcast.

p R-delivered but did not yet A-deliver. Intuitively, these are the messages that were submitted for Atomic Broadcast but are not yet A-delivered, according to p.

In Task 3, process p periodically checks whether $A\_undelivered_p$ contains messages. If so, p enters its next execution of Consensus, say the kth one, by proposing $A\_undelivered_p$ as the next batch of messages to be A-delivered. Process p then waits for the kth Consensus decision, denoted $msgSet^k$. Finally, p A-delivers all the messages in $msgSet^k$ except those it already A-delivered. More precisely, p A-delivers all the messages in the set $A\_deliver_p^k = msgSet^k - A\_delivered_p$, and it does so in some deterministic order that was agreed *a priori* by all processes, e.g., in lexicographical order.

Lemma 13. *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 the agreement property of Reliable Broadcast q eventually R-delivers m, and inserts m into $R\_delivered_q$. □

Lemma 14. *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 $A\_deliver_p^k$, then q eventually A-delivers messages in $A\_deliver_q^k$, and $A\_deliver_p^k = A\_deliver_q^k$.

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 13, 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 $A\_deliver^1_p$, then q eventually A-delivers messages in $A\_deliver^1_q$, and $A\_deliver^1_p = A\_deliver^1_q$. From the algorithm, if p A-delivers messages in $A\_deliver^1_p$, 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*$(1, msgSet^1)$ with the same $msgSet^1$. Since $A\_delivered_p$ and $A\_delivered_q$ are initially empty, and $msgSet^1_p = msgSet^1_q$, we have $A\_deliver^1_p = A\_deliver^1_q$.

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 $\bigcup_{k=1}^{l-1} A\_deliver^k_p$. By the induction hypothesis, $A\_deliver^k_p = A\_deliver^k_q$ for all $1 \leq k \leq l - 1$. So m is not in $\bigcup_{k=1}^{l-1} A\_deliver^k_q$. Since m is in $R\_delivered_p$, by Lemma 13, m is eventually in $R\_delivered_q$. Thus, there is a time after q A-delivers $A\_deliver^{l-1}_q$ 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 $A\_deliver^l_p$, then q A-delivers messages in $A\_deliver^l_q$, and $A\_deliver^l_p = A\_deliver^l_q$. Since p A-delivers messages in $A\_deliver^l_p$, 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*$(l, msgSet^l)$ with the same $msgSet^l$. Note that $A\_deliver^l_p = msgSet^l_p - \bigcup_{k=1}^{l-1} A\_deliver^k_p$, and $A\_deliver^l_q = msgSet^l_q - \bigcup_{k=1}^{l-1} A\_deliver^k_q$. By the induction hypothesis, $A\_deliver^k_p = A\_deliver^k_q$ for all $1 \leq k \leq l - 1$. Since $msgSet^l_p = msgSet^l_q$, $A\_deliver^l_p = A\_deliver^l_q$. □

LEMMA 15. *The algorithm in Figure 7 satisfies the agreement and total order properties of A-broadcast.*

PROOF. Immediate from Lemma 14, and the fact that correct processes A-deliver messages in each batch in the same deterministic order. □

LEMMA 16. *(Validity) If a correct process A-broadcasts m, then it eventually A-delivers m.*

PROOF. The proof is by contradiction. Suppose a correct process p A-broadcasts m but never A-delivers m. By Lemma 15, no correct process A-delivers m.

By Task 1 of Figure 7, p R-broadcasts m. By the validity and agreement properties 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 7 and Lemma 14, there is a k_1, such that for all $l \geq k_1$, 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 $l \geq k_2$. Let $k = max(k_1, k_2)$. Since all correct processes execute $propose(k, -)$, by termination and agreement of Consensus, all correct processes execute $decide(k, msgSet^k)$ with the same $msgSet^k$. By uniform validity of Consensus, some process q executed $propose(k, msgSet^k)$. From our definition of k, q is correct and $msgSet^k$ contains m. Thus all correct processes, including p, A-deliver m—a contradiction that concludes the proof. □

LEMMA 17. *(Uniform integrity) For any message m, each process A-delivers m at most once, and only if m was previously A-broadcast by $sender(m)$.*

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

THEOREM 6. *Consider any system (synchronous or asynchronous) subject to crash failures and where Reliable Broadcast can be implemented. The algorithm in Figure 7 transforms any algorithm for Consensus into an Atomic Broadcast algorithm.*

PROOF. Immediate from Lemmata 15, 16, and 17. □

Since Reliable Broadcast can be implemented in asynchronous systems with crash failures (Section 4), the above theorem shows that Atomic Broadcast is reducible to Consensus in such systems. As we showed earlier, the converse is also true. Thus:[19]

COROLLARY 5. *Consensus and Atomic Broadcast are equivalent in asynchronous systems.*

This equivalence immediately implies that our results regarding Consensus (in particular Corollaries 2 and 4, and Theorem 5) also hold for Atomic Broadcast:

COROLLARY 6. *Atomic Broadcast is solvable using $\mathcal{W}$ in asynchronous systems with $f < n$, and using $\Diamond\mathcal{W}$ in asynchronous systems with $f < \lceil \frac{n}{2} \rceil$.*

COROLLARY 7. *$\Diamond\mathcal{W}_0$ is the weakest failure detector for solving Atomic Broadcast in asynchronous systems with $f < \lceil \frac{n}{2} \rceil$.*

COROLLARY 8. *Atomic Broadcast cannot be solved using $\Diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$.*

Furthermore, Theorem 6 shows that by "plugging in" any *randomised* Consensus algorithm (such as the ones in [Chor and Dwork 1989]) into the algorithm of

[19] All the results stated henceforth in this section are for systems with crash failures.

Figure 7, we automatically get a randomised algorithm for Atomic Broadcast in asynchronous systems.

Corollary 9. *Atomic Broadcast can be solved by randomised algorithms in asynchronous systems with* $f < \lceil \frac{n}{2} \rceil$.

8. COMPARING FAILURE DETECTOR CLASSES

We already saw some relations between the eight failure detector classes that we defined in this paper (Figure 1). In particular, in Section 3 (Corollary 1), we determined that $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$, and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$. This result allowed us to focus on four classes of failure detectors, namely $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, rather than all eight. It is natural to ask whether these four classes (which require Strong Completeness and span the four different types of accuracy) are really distinct or whether some pairs are actually equivalent. More generally, how are $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$ related under the $\succeq$ relation? This section answers these questions.[20]

Clearly, $\mathcal{P} \succeq \mathcal{S}$, $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$, $\mathcal{P} \succeq \diamond\mathcal{P}$, $\mathcal{S} \succeq \diamond\mathcal{S}$, and $\mathcal{P} \succeq \diamond\mathcal{S}$. Are these relations "strict"? For example, it is conceivable that $\mathcal{S} \succeq \mathcal{P}$. If this was true, $\mathcal{P}$ would be equivalent to $\mathcal{S}$ (and the relation $\mathcal{P} \succeq \mathcal{S}$ would not be strict). Also, how are $\mathcal{S}$ and $\diamond\mathcal{P}$ related? Is $\mathcal{S} \succeq \diamond\mathcal{P}$ or $\diamond\mathcal{P} \succeq \mathcal{S}$?

To answer these questions, we begin with some simple definitions. Let $\mathcal{C}$ and $\mathcal{C}'$ be two classes of failure detectors. If $\mathcal{C} \succeq \mathcal{C}'$, and $\mathcal{C}$ is not equivalent to $\mathcal{C}'$, we say that $\mathcal{C}'$ *is strictly weaker than* $\mathcal{C}$, and write $\mathcal{C} \succ \mathcal{C}'$. The following holds:

Theorem 7. $\mathcal{P} \succ \mathcal{S}$, $\diamond\mathcal{P} \succ \diamond\mathcal{S}$, $\mathcal{P} \succ \diamond\mathcal{P}$, $\mathcal{S} \succ \diamond\mathcal{S}$, *and* $\mathcal{P} \succ \diamond\mathcal{S}$. *Furthermore,* $\mathcal{S}$ *and* $\diamond\mathcal{P}$ *are incomparable, i.e., neither* $\mathcal{S} \succeq \diamond\mathcal{P}$ *nor* $\diamond\mathcal{P} \succeq \mathcal{S}$.

The above theorem and Corollary 1 completely characterise the relationship between the eight failure detector classes (defined in Figure 1) under the reducibility relation. Figure 8 illustrates these relations as follows: there is an *undirected edge* between equivalent failure detector classes, and there is a *directed edge* from failure detector class C to class C' if C' is strictly weaker than C.

Even though $\diamond\mathcal{S}$ is strictly weaker than $\mathcal{P}$, $\mathcal{S}$, and $\diamond\mathcal{P}$, it is "strong enough" to solve Consensus and Atomic Broadcast, two powerful paradigms of fault-tolerant computing. This raises an interesting question: Are there any "natural" problems that require classes of failure detectors that are stronger than $\diamond\mathcal{S}$?

To answer this question, consider the problem of *Terminating Reliable Broadcast*, abbreviated here as *TRB* [Hadzilacos and Toueg 1994]. With TRB there is a distinguished process, the *sender* s, that is supposed to broadcast a single message from a set $\mathcal{M}$ of possible messages. TRB is similar to Reliable Broadcast, except that it requires that every correct process *always* deliver a message — even if the sender s is faulty and, say, crashes before broadcasting. For this requirement to be satisfiable, processes must be allowed to deliver a message that was not actually broadcast. Thus, TRB allows the delivery of a special message $F_s \notin \mathcal{M}$ which states that the sender s is faulty (by convention, $sender(F_s) = s$).

With TRB for sender s, s can broadcast any message $m \in \mathcal{M}$, processes can deliver any message $m \in \mathcal{M} \cup \{F_s\}$, and the following hold:

[20]The results presented here are not central to this paper, hence the proofs are omitted.

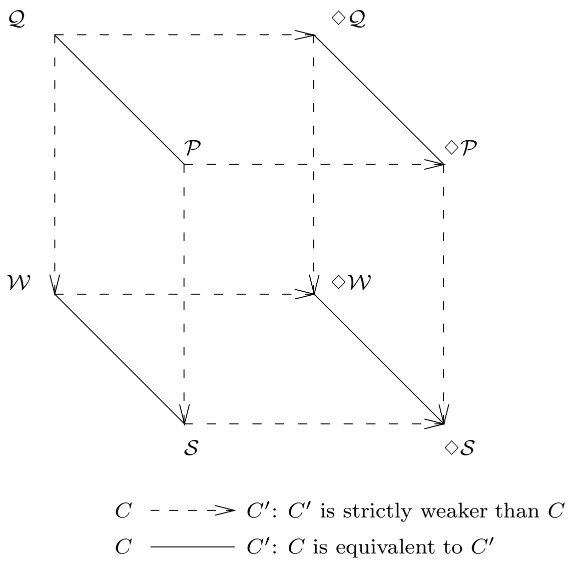

Fig. 8. Comparing the eight failure detector classes by reducibility.

Termination. Every correct process eventually delivers exactly one message.

Validity. If s is correct and broadcasts a message m, then it eventually delivers m.

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

Integrity. If a correct process delivers a message m then $sender(m) = s$. Furthermore, if $m \neq F_s$ then m was previously broadcast by s.

The reader should verify that the specification of TRB for sender s implies that a correct process delivers the special message F_s only if s is indeed faulty.

TRB is a well-known and studied problem, usually known under the name of the *Byzantine Generals' Problem* [Pease et al. 1980; Lamport et al. 1982].[21] It turns out that in order to solve TRB in asynchronous systems one needs to use the strongest class of failure detectors that we defined in this paper. Specifically:

THEOREM 8.

(1) TRB can be solved using $\mathcal{P}$ in asynchronous systems with any number of crashes.

(2) TRB cannot be solved using either $\mathcal{S}$, $\Diamond\mathcal{P}$, or $\Diamond\mathcal{S}$ in asynchronous systems. This impossibility result holds even under the assumption that at most one crash may occur.

[21] We refrain from using this name because it is often associated with *Byzantine failures*, while we consider only crash failures here.

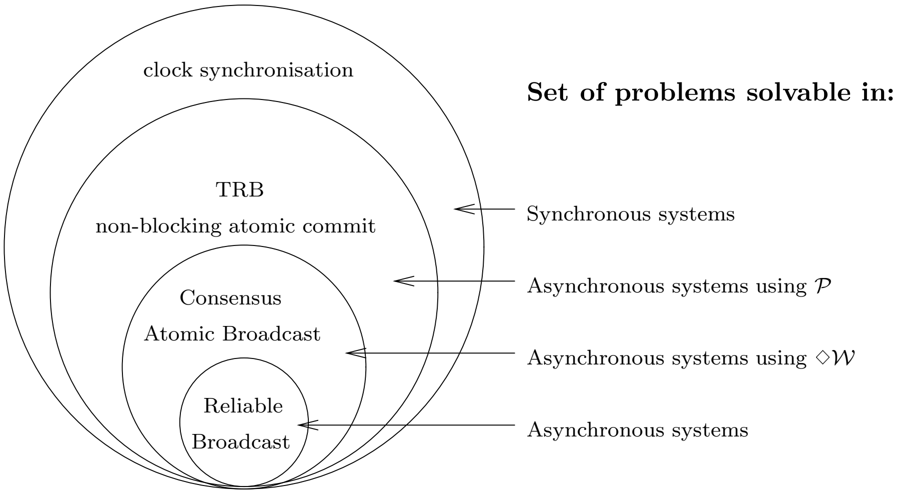

Fig. 9. Problem solvability in different distributed computing models.

In fact, $\mathcal{P}$ is the weakest failure detector class that can be used to solve repeated instances of TRB (multiple instances for each process as the distinguished sender).

TRB is not the only "natural" problem that can be solved using $\mathcal{P}$ but cannot be solved using $\Diamond\mathcal{W}$. Other examples include the *non-blocking atomic commitment problem* [Chandra and Larrea 1994; Guerraoui 1995], and a form of *leader election* [Sabel and Marzullo 1995]. Figure 9 summarises these results.

9. RELATED WORK

9.1 Partial synchrony

Fischer, Lynch and Paterson showed that Consensus cannot be solved in an asynchronous system subject to crash failures [Fischer et al. 1985]. 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 paper.

A different tack on circumventing the unsolvability of Consensus is pursued in [Dolev et al. 1987] and [Dwork et al. 1988]. 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, [Dolev et al. 1987] define 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. [Dolev et al. 1987] identify 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, [Dolev et al. 1987] delineate 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?".

[Dwork et al. 1988] consider two models of partial synchrony. Roughly speaking, the first model (denoted $\mathcal{M}_1$ here) stipulates that in every execution there are bounds on relative process speeds and on message transmission times, but these bounds are not known. In the second model (denoted $\mathcal{M}_2$) these bounds are known, but they hold only after some unknown time (called *GST* for *Global Stabilisation Time*). In each one of these two models (with crash failures), it is easy to implement an Eventually Perfect failure detector $\mathcal{D} \in \diamond\mathcal{P}$. In fact, we can implement such a failure detector in a weaker model of partial synchrony (denoted $\mathcal{M}_3$): one in which bounds exist but they are not known *and* they hold only after some unknown time GST.[22] Since $\diamond\mathcal{P} \succeq \diamond\mathcal{W}$, by Corollaries 3 and 6, this implementation immediately gives Consensus and Atomic Broadcast solutions for $\mathcal{M}_3$ and, a fortiori, for $\mathcal{M}_1$ and $\mathcal{M}_2$.

The implementation of $\mathcal{D} \in \diamond\mathcal{P}$ for $\mathcal{M}_3$, which uses an idea found in [Dwork et al. 1988], works as follows (see Figure 10). To measure elapsed time, each process p maintains a local clock, say by counting the number of steps that it takes. 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 $\Delta_p(q)$ time units on its clock, 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 its time-out period $\Delta_p(q)$.

Theorem 9. *Consider a partially synchronous system $\mathcal{S}$ that conforms to $\mathcal{M}_3$, i.e., for every run of $\mathcal{S}$ there is a* Global Stabilisation Time *(GST) after which some bounds on relative process speeds and message transmission times hold (the values of GST and these bounds are* not *known). The algorithm in Figure 10 implements an Eventually Perfect failure detector $\mathcal{D} \in \diamond\mathcal{P}$ in $\mathcal{S}$.*

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,

[22]Note that every system that conforms to $\mathcal{M}_1$ or $\mathcal{M}_2$ also conforms to $\mathcal{M}_3$.

Every process p executes the following:

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

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

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

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

Fig. 10. A time-out based implementation of $\mathcal{D} \in \Diamond\mathcal{P}$ in models of partial synchrony.

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 removed from $output_p$ only in Task 3, and every time this occurs p's 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 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. $\square$

In this paper we have not considered communication failures. In the second model of partial synchrony of [Dwork et al. 1988], where bounds are known but hold only after GST, messages sent before GST can be lost. We now re-define $\mathcal{M}_2$ and $\mathcal{M}_3$

analogously — messages that are sent before GST can be lost — and examine how this affects our results so far.[23] The failure detector algorithm in Figure 10 still implements an Eventually Perfect failure detector $\mathcal{D} \in \diamond\mathcal{P}$ in $\mathcal{M}_3$, despite initial message losses now allowed by this model. On the other hand, these initial message losses invalidate the Consensus algorithm in Figure 6. It is easy to modify this algorithm, however, so that it does work in $\mathcal{M}_3$: One can adopt the techniques used in [Dwork et al. 1988] to mask the loss of messages that are sent before GST.

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 parameters that define $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, or the five parameters considered in [Dolev et al. 1987]), we can consider the *axiomatic properties* that failure detectors must have in order to solve Consensus. The problem of implementing a certain type of 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 a certain type of failure detector we show that Consensus is solvable in *all* systems in which this type of failure detector can be implemented. An algorithm that relies on the axiomatic properties of a 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 this 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 [Dolev et al. 1987], which identified a *set* of minim*al* models of partial synchrony in which Consensus is solvable, [Chandra et al. 1992] exhibit a *single* minim*um* failure detector, $\diamond\mathcal{W}_0$, that can be used to solve Consensus. The technical device that makes this possible is the notion of *reduction* between failure detectors.

9.2 Unreliable failure detection in shared memory systems

Loui and Abu-Amara showed that in asynchronous shared memory systems with atomic read/write registers, Consensus cannot be solved even if at most one process may crash [Loui and Abu-Amara 1987].[24] This raises the following question: can we use unreliable failure detectors to circumvent this impossibility result?

Lo and Hadzilacos [Lo and Hadzilacos 1994] showed that this is indeed possible: They gave an algorithm that solves Consensus using $\diamond\mathcal{W}$ (in shared memory systems with registers). This algorithm tolerates *any* number of faulty processes — in contrast to our result showing that in message-passing systems $\diamond\mathcal{W}$ can be used to solve Consensus only if there is a majority of correct processes. Recently, Neiger extended the work of Lo and Hadzilacos by studying the conditions under which unreliable failure detectors boost the Consensus power of shared objects [Neiger

[23] Note that model $\mathcal{M}_3$ is now *strictly* weaker than models $\mathcal{M}_1$ and $\mathcal{M}_2$: there exist systems that conform to $\mathcal{M}_3$ but not to $\mathcal{M}_1$ or $\mathcal{M}_2$.

[24] The proof in [Loui and Abu-Amara 1987] is similar to the proof that Consensus is impossible in message-passing systems when send and receive are not part of the same atomic step [Dolev et al. 1987].

1995].

9.3 The Isis toolkit

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 [Ricciardi and Birman 1991]. 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.[25] 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, premature 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 given failure detector "malfunctions", some messages may be delayed, but no message is ever delivered out of order, and no correct process is forced to crash. If the failure detector stops malfunctioning, outstanding messages are eventually delivered. Thus, we can set time-out periods more aggressively than a system like Isis: in practice, we would set our failure detector time-out periods closer to the average case, while systems like Isis must set time-outs closer to the worst-case.

9.4 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 [Attiya et al. 1991], which, as pointed out by the authors, "can be viewed as an asynchronous algorithm that uses a fault detection (e.g., timeout) mechanism."

Recent work shows that the *Group Membership* problem cannot be solved in

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

asynchronous systems with crash failures, even if one adopts the Isis approach of crashing processes that are suspected to be faulty but are actually correct [Chandra et al. 1995]. As with Consensus and Atomic Broadcast, this impossibility result can be circumvented by the addition of unreliable failure detectors.

ACKNOWLEDGMENTS

We are deeply grateful to Vassos Hadzilacos for his crucial help in revising this paper. The comments and suggestions of the anonymous referees, Navin Budhiraja, and Bernadette Charron-Bost, were also instrumental in improving the paper. Finally, we would like to thank Prasad Jayanti for greatly simplifying the algorithm in Figure 3.

APPENDIX

A hierarchy of failure detector classes and bounds on fault-tolerance

In the preceding sections, 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 classes that will be used to unify some of our results, and to refine the lower bound on fault-tolerance given in Section 6.3. This infinite hierarchy consists of a continuum of repentant failure detectors ordered by the maximum number of mistakes that each one can make.

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 at process p about process q* if at time t, p begins to suspect that q has crashed even though $q \notin F(t)$. Formally:

$$[q \notin F(t), q \in H(p,t)] \text{ and } [q \notin H(p,t-1)]$$

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 at p. The continuous *retention* of q into $\mathcal{D}_p$ does not count as additional mistakes. Thus, a failure detector can make multiple mistakes at 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 (it may, however, continue to give incorrect information).

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 (it may, however, continue to give incorrect information even 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 $n(n-1)$ 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 [Halpern and Moses 1990]. 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

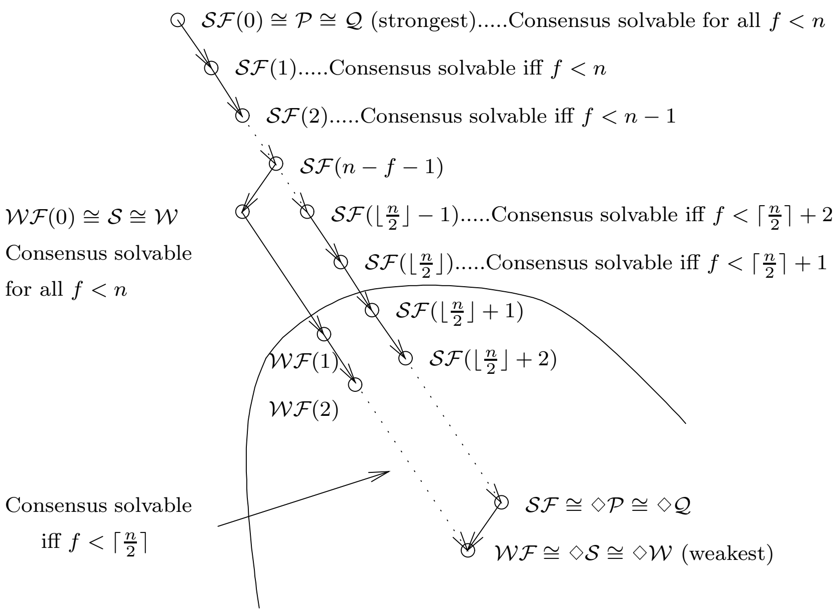

Fig. 11. Classes of repentant failure detectors ordered by reducibility. For each class $\mathcal{C}$, the maximum number of faulty processes for which Consensus can be solved using $\mathcal{C}$ is given.

reader should refer to the seminal work done in [Moses et al. 1986; Halpern and Moses 1990].

Recall that in Section 2.2 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.

We now informally define a hierarchy of repentant failure detectors that differ by 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 2.2.

A hierarchy of repentant failure detectors

Consider the failure detectors that satisfy weak completeness, one of the four types of accuracy that we defined in the previous section, and repentance. These failure detectors can be grouped into four classes according to the actual accuracy property that they satisfy:

$\mathcal{SF}(k)$: the class of *Strongly k-Mistaken failure detectors*,

$\mathcal{SF}$: the class of *Strongly Finitely Mistaken failure detectors*,

$\mathcal{WF}(k)$: the class of *Weakly k-Mistaken failure detectors*, and

$\mathcal{WF}$: the class of *Weakly Finitely Mistaken failure detectors.*

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 any failure detector $\mathcal{D} \in \mathcal{SF}((n-f)-1)$ makes fewer mistakes than the number of correct processes, there is at least one correct process that $\mathcal{D}$ never suspects. Thus, $\mathcal{D}$ is also weakly 0-mistaken, and we conclude that $\mathcal{SF}((n-f)-1) \succeq \mathcal{WF}(0)$. Furthermore, it is clear that $\mathcal{SF} \succeq \mathcal{WF}$.

These classes of repentant failure detectors can be ordered by reducibility into an infinite hierarchy, which is illustrated in Figure 11 (an edge $\rightarrow$ represents the $\succeq$ relation). Each failure detector class defined in Section 2.4 is equivalent to some class in this hierarchy. In particular, it is easy to show that:

Observation 2. $\mathcal{P} \cong \mathcal{Q} \cong \mathcal{SF}(0)$, $\mathcal{S} \cong \mathcal{W} \cong \mathcal{WF}(0)$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q} \cong \mathcal{SF}$, and $\diamond\mathcal{S} \cong \diamond\mathcal{W} \cong \mathcal{WF}$.

For example, it is easy to see that the algorithm in Figure 3 transforms any failure detector in $\mathcal{WF}$ into one in $\diamond\mathcal{W}$. Other conversions are similar or straightforward and are therefore omitted. Note that $\mathcal{P}$ and $\diamond\mathcal{W}$ are the strongest and weakest failure detector classes in this hierarchy, respectively. From Corollaries 2 and 6, and Observation 2 we have:

Corollary 10. *Consensus and Atomic Broadcast are solvable using* $\mathcal{WF}(0)$ *in asynchronous systems with* $f < n$.

Similarly, from Corollaries 3 and 6, and Observation 2 we have:

Corollary 11. *Consensus and Atomic Broadcast are solvable using* $\mathcal{WF}$ *in asynchronous systems with* $f < \lceil \frac{n}{2} \rceil$.

Tight bounds on fault-tolerance

Since Consensus and Atomic Broadcast are equivalent in asynchronous systems with any number of faulty processes (Corollary 5), we can focus on establishing fault-tolerance bounds for Consensus. In Section 6, we showed that failure detectors with *perpetual* accuracy (i.e., in $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, 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., in $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, 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 class $\mathcal{C}$ in our infinite hierarchy, and determining how many correct processes are necessary to solve Consensus using $\mathcal{C}$. The results are illustrated in Figure 11.

There are two cases depending on whether we assume that the system has a majority of correct processes or not. If a majority of the processes are correct, Consensus can be solved with $\diamond\mathcal{W}$, the weakest failure detector class in the hierarchy. Thus:

Observation 3. In asynchronous systems with $f < \lceil \frac{n}{2} \rceil$, Consensus can be solved using any failure detector class in the hierarchy of Figure 11.

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 10. *In asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$, if $m > n - f$ then Consensus cannot be solved using $\mathcal{SF}(m)$.*

PROOF *(sketch).* Consider an asynchronous system with $f \geq \lceil \frac{n}{2} \rceil$ and assume $m > n - f$. We show that there is a failure detector $\mathcal{D} \in \mathcal{SF}(m)$ such that no algorithm solves Consensus using $\mathcal{D}$. We do so by describing the behaviour of a Strongly m-Mistaken failure detector $\mathcal{D}$ such that for every algorithm A, there is a run R_A of A using $\mathcal{D}$ that violates the specification of Consensus.

Since $1 \leq n - f \leq \lfloor \frac{n}{2} \rfloor$, 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. Henceforth, we only consider runs in which all processes in $\Pi_{crashed}$ crash at the beginning of the run. Let $q_0 \in \Pi_0$ and $q_1 \in \Pi_1$. Consider the following two runs of A using $\mathcal{D}$:

Run $R_0 = \langle F_0, H_0, I_0, S_0, T_0 \rangle$. All processes propose 0. 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 $q_0 \in \Pi_0$ 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., $\forall t \in \mathcal{T}, \forall q \neq q_0 : H_0(q, t) = \emptyset$. Clearly, $\mathcal{D}$ satisfies the specification of a Strongly m-Mistaken failure detector in R_0.

Run $R_1 = \langle F_1, H_1, I_1, S_1, T_1 \rangle$. All processes 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 $q_1 \in \Pi_1$ permanently suspects every process in $\Pi_0 \cup \Pi_{crashed}$, and no other process suspects any process. $\mathcal{D}$ satisfies the specification of a Strongly m-Mistaken failure detector in R_1.

If R_0 or R_1 violates the specification of Consensus, A does not solve Consensus using $\mathcal{D}$, as we wanted to show. Now assume that both R_0 and R_1 satisfy the specification of Consensus. In this case, all correct processes decide 0 in R_0 and 1 in R_1. Let t_0 be the time at which q_0 decides 0 in R_0, and let t_1 be the time at which q_1 decides 1 in R_1. We now describe the behaviour of $\mathcal{D}$ and a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of A using $\mathcal{D}$ that violates the specification of Consensus.

In R_A all processes in Π_0 propose 0 and all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1. 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, (i) $\mathcal{D}$ behaves as in R_0, and (ii) all the processes in Π_1 are "very slow": they do not take any steps. Thus, until time t_0, no process in Π_0 can distinguish between R_0 and R_A, and all processes in Π_0 execute exactly as in R_0. In particular,

q_0 decides 0 at time t_0 in R_A (as it did in R_0). Note that by time t_0, $\mathcal{D}$ 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 behaviour of $\mathcal{D}$ and run R_A continue as follows:

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

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

(3) 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). Since q_0 previously decided 0, R_A violates the agreement property of Consensus.

(4) From time t_0+t_1 onwards, no more processes crash and every correct process suspects exactly all the processes that have crashed. Thus, $\mathcal{D}$ satisfies weak completeness, repentance, and makes no further mistakes.

By (2) and (4), $\mathcal{D}$ satisfies the specification of a Strongly m-Mistaken failure detector, i.e., $\mathcal{D} \in \mathcal{SF}(m)$. From (3), A does not solve Consensus using $\mathcal{D}$. □

We now show that the above lower bound is tight:

Theorem 11. *In asynchronous systems with $m \le n-f$, Consensus can be solved using $\mathcal{SF}(m)$.*

Proof. Suppose $m < n-f$, and consider any failure detector $\mathcal{D} \in \mathcal{SF}(m)$. Since m, the number of mistakes made by $\mathcal{D}$, is less than the number of correct processes, there is at least one correct process that $\mathcal{D}$ never suspects. Thus, $\mathcal{D}$ satisfies weak accuracy. By the definition of $\mathcal{SF}(m)$, $\mathcal{D}$ also satisfies weak completeness. So $\mathcal{D} \in \mathcal{W}$, and it can be used to solve Consensus (Corollary 2).

Suppose $m = n-f$. Even though $\mathcal{D}$ can now make a mistake about *every* correct process, it can still be used to solve Consensus (even if a majority of the processes are faulty). The corresponding algorithm uses rotating coordinators, and is similar to the one for $\Diamond\mathcal{W}$ given in Figure 6. Because of this similarity, we omit the details. □

From the above two theorems:

Corollary 12. *In asynchronous systems with $f \ge \lceil \frac{n}{2} \rceil$, Consensus can be solved using $\mathcal{SF}(m)$ if and only if $m \le n-f$.*

We now turn our attention to solving Consensus using $\mathcal{WF}(m)$.

Theorem 12. *In asynchronous systems with $f \ge \lceil \frac{n}{2} \rceil$, Consensus cannot be solved using $\mathcal{WF}(m)$ with $m > 0$.*

Proof. In Theorem 10, we described a failure detector $\mathcal{D}$ that cannot be used to solve Consensus in asynchronous systems with $f \ge \lceil \frac{n}{2} \rceil$. It is easy to verify that $\mathcal{D}$ makes at most one mistake about each correct process, and thus $\mathcal{D} \in \mathcal{WF}(1)$. □

From Corollary 10 and the above theorem, we have:

COROLLARY 13. *In asynchronous systems with* $f \geq \lceil \frac{n}{2} \rceil$*, Consensus can be solved using* $\mathcal{WF}(m)$ *if and only if* $m = 0$*.*

References

AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. 1991 . Transis: A communication sub-system for high availability. Technical Report CS91-13 (Nov.), Computer Science Department, The Hebrew University of Jerusalem.

ATTIYA, H., BAR-NOY, A., DOLEV, D., KOLLER, D., PELEG, D., AND REISCHUK, R. 1987 . Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science* (Oct. 1987), pp. 337–346. IEEE Computer Society Press.

ATTIYA, H., DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1991 . 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), pp. 359–369. ACM Press.

BEN-OR, M. 1983 . Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (Aug. 1983), pp. 27–30. ACM Press.

BERMAN, P., GARAY, J. A., AND PERRY, K. J. 1989 . Towards optimal distributed consensus. In *Proceedings of the Thirtieth Symposium on Foundations of Computer Science* (Oct. 1989), pp. 410–415. IEEE Computer Society Press.

BIRAN, O., MORAN, S., AND ZAKS, S. 1988 . 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* (Aug. 1988), pp. 263–275. ACM Press.

BIRMAN, K. P., COOPER, R., JOSEPH, T. A., KANE, K. P., AND SCHMUCK, F. B. 1990 . *Isis - A Distributed Programming Environment.*

BIRMAN, K. P. AND JOSEPH, T. A. 1987 . Reliable communication in the presence of failures. *ACM Transactions on Computer Systems 5*, 1 (Feb.), 47–76.

BRACHA, G. AND TOUEG, S. 1985 . Asynchronous consensus and broadcast protocols. *Journal of the ACM 32*, 4 (Oct.), 824–840.

BRIDGLAND, M. AND WATRO, R. 1987 . Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing* (August 1987), pp. 52–63. ACM Press.

BUDHIRAJA, N., GOPAL, A., AND TOUEG, S. 1990 . Early-stopping distributed bidding and applications. In *Proceedings of the Fourth International Workshop on Distributed Algorithms* (Sept. 1990), pp. 301–320. Springer-Verlag.

CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1992 . The weakest failure detector for solving consensus. Technical Report 92-1293 (July), Department of Computer Science, Cornell University. Available from `ftp://ftp.cs.cornell.edu/pub/chandra/failure.detectors.weakest.dvi.Z`. A preliminary version appeared in the *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 147–158. ACM Press, August 1992.

CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1995 . Impossibility of group membership in asynchronous systems. Technical Report 95-1533 (August), Computer Science Department, Cornell University, Ithaca, New York 14853.

CHANDRA, T. D. AND LARREA, M. 1994 . E-mail correspondence. Showed that $\Diamond\mathcal{W}$ cannot be used to solve non-blocking atomic commit.

CHANDRA, T. D. AND TOUEG, S. 1990 . Time and message efficient reliable broadcasts. In *Proceedings of the Fourth International Workshop on Distributed Algorithms* (Sept. 1990), pp. 289–300. Springer-Verlag.

CHANG, J. AND MAXEMCHUK, N. 1984 . Reliable broadcast protocols. *ACM Transactions on Computer Systems 2*, 3 (Aug.), 251–273.

CHOR, B. AND DWORK, C. 1989. Randomization in byzantine agreement. *Advances in Computer Research 5*, 443–497.

CRISTIAN, F. 1987. Issues in the design of highly available computing services. In *Annual Symposium of the Canadian Information Processing Society* (July 1987), pp. 9–16. Also IBM Research Report RJ5856, July 1987.

CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. 1985. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing* (June 1985), pp. 200–206. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

CRISTIAN, F., DANCEY, R. D., AND DEHN, J. 1990. Fault-tolerance in the advanced automation system. Technical Report RJ 7424 (April), IBM Research Laboratory.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34*, 1 (Jan.), 77–97.

DOLEV, D., LYNCH, N. A., PINTER, S. S., STARK, E. W., AND WEIHL, W. E. 1986. Reaching approximate agreement in the presence of faults. *Journal of the ACM 33*, 3 (July), 499–516.

DWORK, C., LYNCH, N. A., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (April), 288–323.

FISCHER, M. J. 1983. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273 (June), Department of Computer Science, Yale University.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (April), 374–382.

GOPAL, A., STRONG, R., TOUEG, S., AND CRISTIAN, F. 1990. Early-delivery atomic broadcast. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing* (Aug. 1990), pp. 297–310. ACM Press.

GUERRAOUI, R. 1995. Revisiting the relationship between non blocking atomic commitment and consensus. In *Proceedings of the Ninth International Workshop on Distributed Algorithms* (September 1995), pp. 87–100. Springer-Verlag.

HADZILACOS, V. AND TOUEG, S. 1993. Fault-tolerant broadcasts and related problems. In S. J. MULLENDER Ed., *Distributed Systems*, Chapter 5, pp. 97–145. Addison-Wesley.

HADZILACOS, V. AND TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425 (May), Computer Science Department, Cornell University, Ithaca, New York 14853. Available by anonymous ftp from `ftp://ftp.db.toronto.edu/pub/vassos/fault.tolerant.broadcasts.dvi.Z`. An earlier version is also available in [Hadzilacos and Toueg 1993].

HALPERN, J. Y. AND MOSES, Y. 1990. Knowledge and common knowledge in a distributed environment. *Journal of the ACM 37*, 3 (July), 549–587.

LAMPORT, L. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks 2*, 95–114.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. on Programm. Lang. Syst. 4*, 3 (July), 382–401.

LO, W. K. AND HADZILACOS, V. 1994. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the Eighth International Workshop on Distributed Algorithms* (Sept. 1994), pp. 280–295. Springer-Verlag. Available from `ftp://ftp.db.toronto.edu/pub/vassos/failure.detectors.shared.memory.ps.Z`.

LOUI, M. AND ABU-AMARA. 1987. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research 4*, 163–183.

MOSES, Y., DOLEV, D., AND HALPERN, J. Y. 1986. Cheating husbands and other stories: a case study of knowledge, action, and communication. *Distributed Computing 1*, 3, 167–176.

MULLENDER, S. J. Ed. 1987. *The Amoeba distributed operating system: Selected papers 1984 - 1987*. Centre for Mathematics and Computer Science.

NEIGER, G. 1995. Failure detectors and the wait-free hierarchy. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing* (Aug. 1995), pp. 100–109. ACM Press.

NEIGER, G. AND TOUEG, S. 1990 . Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms 11*, 3 (Sept.), 374–419.

PEASE, M., SHOSTAK, R., AND LAMPORT, L. 1980 . Reaching agreement in the presence of faults. *Journal of the ACM 27*, 2 (April), 228–234.

PETERSON, L. L., BUCHOLZ, N. C., AND SCHLICHTING, R. D. 1989 . Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems 7*, 3 (Aug.), 217–246.

PITTELLI, F. AND GARCIA-MOLINA, H. 1989 . Reliable scheduling in a tmr database system. *ACM Transactions on Computer Systems 7*, 1 (Feb.), 25–60.

POWELL, D. Ed. 1991 . *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag.

REISCHUK, R. 1982 . A new solution for the Byzantine general's problem. Technical Report RJ 3673 (Nov.), IBM Research Laboratory.

RICCIARDI, A. AND BIRMAN, K. P. 1991 . Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing* (Aug. 1991), pp. 341–351. ACM Press.

SABEL, L. AND MARZULLO, K. 1995 . Election vs. consensus in asynchronous systems. Technical Report TR95-411 (Feb.), University of California at San Diego. Available at `ftp://ftp.cs.cornell.edu/pub/sabel/tr94-1413.ps`.

SCHNEIDER, F. B. 1990 . Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (Dec.), 299–319.

WENSLEY, J. H., LAMPORT, L., GOLDBERG, J., GREEN, M. W., LEVITT, K. N., MELLIAR-SMITH, P., SHOSTAK, R. E., AND WEINSTOCK, C. B. 1978 . SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE 66*, 10 (Oct.), 1240–1255.