Unreliable Failure Detectors for Asynchronous Systems*

(Preliminary Version)

Tushar Deepak Chandra and Sam Toueg

Department of Computer Science
Upson Hall, Cornell University
Ithaca, New York 14853

chandra,sam@cs.cornell.edu

Jul 23, 1991

Abstract

We introduce the concept of *failure detectors* for asynchronous systems with crash failures. We show that even with a failure detector that makes an unbounded and possibly infinite number of mistakes, we can solve the Consensus and Atomic Broadcast problems, two fundamental paradigms of fault-tolerant computing that are known to be unsolvable in asynchronous systems. We characterize failure detectors in terms of their *completeness* and *accuracy* properties, and classify them in a hierarchy ordered by a reducibility relation. We present matching upper and lower bounds on the fault-tolerance of solutions to Consensus and Atomic Broadcast for members of this hierarchy.

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames Grant NAG-2-593, and in part by Grants from IBM and Siemens Corp.

1 Introduction

The asynchronous model of distributed computing has been extensively studied. Informally, an *asynchronous distributed system* is one in which message transmission times and relative processor speeds are both unbounded. Thus an algorithm designed for an asynchronous system does not rely on such bounds for its correctness. In practice, asynchrony is introduced by unpredictable loads on the system.

Although the asynchronous model of computation is attractive for the reasons outlined above, it is well known that many fundamental paradigms of fault-tolerant distributed computing that are solvable in synchronous systems, are unsolvable in asynchronous systems. In particular, it is well known that *Consensus*, and several forms of reliable broadcast, including *Atomic Broadcast*, cannot be solved deterministically in an asynchronous system that is subject to even a single *crash* failure [FLP85,DDS87].

To circumvent these impossibility results, previous research focused on the use of randomization techniques [CD89], the definition of some weaker problems and their solutions[DLP$^+$86, ABND$^+$87,BW87], or the study of several models of *partial synchrony* [DDS87,DLS88]. However, 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 of an external failure detection mechanism. In particular, we model the concept of *failure detectors* for systems with *crash* failures.

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. The local modules at two different processes may have different lists of suspects. 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.

Our failure detectors are characterized by two types of properties: *completeness* and *accuracy*. Informally, *completeness* requires that the failure detector eventually suspects every process that actually crashes[1], while *accuracy* restricts the mistakes that a failure detector can make. We consider several types of failure detectors that differ by the strength of their *accuracy* requirement.

By defining failure detectors in terms of *abstract* properties as opposed to giving specific *implementations*[2], we can 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). 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. An example of such a system is the *ISIS Toolkit* [BCJ$^+$90] for asynchronous fault-tolerant distributed computing.

Given a failure detector with a very weak type of accuracy, we are able to solve both Consensus and Atomic Broadcast in asynchronous systems. This failure detector is allowed to make an *unbounded* and possibly *infinite* number of mistakes: The failure detector can repeatedly add and then remove *correct* processes from its list of suspects (this reflects the inherent difficulty of determining whether a process or a link is just slow or whether it has crashed). Its only accuracy requirement is that there is *one* process p among the correct processes such that, *eventually*, the failure detector stops suspecting p. However, this weak failure detector never indicates which process p is actually correct. Furthermore, processes never know when the failure detector will stop suspecting some correct process. We believe that the weak accuracy requirement of this failure detector is achievable in most practical systems (a possible implementation is described later in this introduction).

It is important to note that even though our failure detectors can erroneously suspect that a correct process p has crashed, p is still required to behave like every other correct process in the system. In particular, p is *not* exempt from satisfying the complete problem specification as it applies to correct processes. For example, our solution to the Atomic Broadcast problem in asynchronous systems has the following desirable properties. Suppose the failure detector *continuously* suspects that a process q has crashed. If q is actually correct, then, despite the permanent mistake of the failure detector, q must still deliver the same set of messages, in the same order, as the other correct processes. Furthermore, whenever q broadcasts a message m, all correct processes must eventually deliver m.

We define a *reducibility* relation on failure detectors, and order failure detectors into a hierarchy according to this relation. We show that certain failure detectors in this hierarchy can be used to solve Consensus and Atomic Broadcast if and only if $f < \frac{n}{2}$ (where f is the maximum

[1] 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.

[2] The hardware or software implementation of failure detectors is not the concern of this paper.

number of processes that may crash, and n is the number of processes in the system). For other failure detectors in the hierarchy, we describe solutions to Consensus and Atomic Broadcast that only require $f < n$. Furthermore, we show exactly where in the hierarchy the requirement of $f < \frac{n}{2}$ becomes necessary.

We now outline a possible implementation of the weak type of failure detector that is sufficient to solve Consensus and Atomic Broadcast in practical systems. This particular implementation is based on time-outs (see Appendix B for a detailed description). Informally, if a process times-out on some process q, it adds q to its list of suspects, and it broadcasts a message to all processes (including q) with this information. Any process that receives this broadcast adds q to its list of suspects. If q has not crashed, it broadcasts a refutation. If a process receives q's refutation, it removes q from its list of suspects. Note that because the underlying system is asynchronous, an *unbounded* sequence of *premature* time-outs with corresponding refutations can indeed occur, and this causes processes like q to be repeatedly added and then removed from the list of suspects. However, it is reasonable to assume that in any *practical* asynchronous system one can design a time-out mechanism that is not *totally* arbitrary: there is at least one correct process p such that *eventually* there are no more premature time-outs on p[3]. This gives the required *accuracy* property, i.e., there is a time after which p is permanently removed from the list of processes suspected to have crashed. It is also reasonable to assume that after a process q actually crashes, some correct process will eventually time-out on q (and broadcast its suspicion that q has crashed). Thus every correct process eventually adds q to its list of suspects. Since q crashed, it cannot initiate the refutation mechanism described above, so q remains on the list of suspects of every correct process. This gives the failure detector's *completeness* property.

In summary, the main contributions of this paper are as follows. We introduce and formalize the concept of failure detectors in terms of desirable *properties* (rather than in terms of implementations). We show that a very weak type of failure detector is sufficient to solve fundamental paradigms of fault-tolerant computing, such as Consensus and Atomic Broadcast, in asynchronous distributed systems. As we argued above, this failure detector can be implemented in most *practical* asynchronous systems[4]. Thus, our results bridge the gap between known theoretical impossibility results, and the need for practical fault-tolerant solutions in existing asynchronous systems.

[3] Actually, it is sufficient for time-outs on p to stop only long enough for the intended application to complete. Note that in most systems, premature time-outs are likely to occur because of *temporary* network overloads and failures.

[4] However, even our weakest failure detector cannot be implemented in the classical model of asynchronous computation: This paper shows that such an implementation can be used to solve Consensus, which is known to be unsolvable in this model [FLP85].

The remainder of this paper is organized as follows. In Section 2, we briefly describe our model. In Section 3, we describe six classes of failure detectors in terms of their properties. In Section 4, we present a simple broadcast primitive that is repeatedly used in our results. In Section 5 we define the Consensus problem. In Sections 6 and 7, we describe solutions to the Consensus problem with several types of failure detectors. In Section 8, we show how to solve Atomic Broadcast. Section 9 discusses related work. In Appendix A, we define a failure detector hierarchy and derive a lower bound on fault-tolerance. In Appendix B we briefly describe a timeout-based implementation of a weak type of failure detector.

2 System Model

We consider *asynchronous* distributed systems where relative processor speeds and message transmission times are finite but unbounded. We assume that every pair of processes is connected by a reliable communication channel. Processes can fail by *crashing*, i.e. by prematurely halting[5].

Informally, a *run* is an infinite execution of the system. Given any run σ, $crashed(t, \sigma)$ is the set of processes that have crashed by time t in σ, and $up(t, \sigma) = Q - crashed(t, \sigma)$, where Q is the set of all the processes in the system. Let $crashed(\sigma) = \bigcup_t crashed(t, \sigma)$ and $up(\sigma) = Q - crashed(\sigma)$.

If $p \in up(\sigma)$ we say that p is *correct* in σ. Otherwise, $p \in crashed(\sigma)$, and we say that p is *faulty* in σ. The reference to σ is omitted when it is unambiguous.

3 Failure Detectors: Model and Properties

We define a failure detector to be a distributed oracle—each process has its own local failure detector module, and the failure detector modules of two different processes need not agree on the list of processes that are suspected to have crashed[6]. Let $Q = \{p_1, p_2, ..., p_n\}$. We define a *distributed failure detector* D as the vector $D = \langle D_{p_1}, D_{p_2}, ..., D_{p_n} \rangle$, where D_p, the failure detector module at process p, is a function from time and the set of all runs to 2^Q. $D_p(t, \sigma)$ is the set of processes that are suspected to have crashed by p's failure detector module at time t in run σ. After a process crashes we don't care what its failure detector module indicates. We formalize this by assuming that if $p \in crashed(t, \sigma)$ then $D_p(t, \sigma) = \emptyset$. If $q \in D_p(t, \sigma)$, we say that p suspects q at time t in run σ.

[5] Process *recovery* is considered in the full version of this paper, but is omitted in this preliminary version.

[6] For example, each process may have its own independent time-out mechanism.

We characterize failure detectors by the *completeness* and *accuracy* properties that they must satisfy. These properties are defined in the next three sections.

3.1 Completeness

We consider two types of *completeness* properties:

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

$$\forall \sigma, \forall p \in crashed(\sigma), \forall q \in up(\sigma), \exists t, \forall t' \geq t : p \in D_q(t', \sigma)$$

- *Weak completeness*: Eventually every process that crashes is permanently suspected by *some* correct process[7].

$$\forall \sigma, \forall p \in crashed(\sigma), \exists q \in up(\sigma), \exists t, \forall t' \geq t : p \in D_q(t', \sigma)$$

However, *completeness* by itself is not a useful property: a failure detector may trivially satisfy this property by always suspecting *all* the processes in the system. To preclude such behavior, a failure detector must also satisfy an *accuracy* requirement that restricts the *mistakes* that a failure detector can make. Sections 3.2 and 3.3 consider *accuracy* and *eventual accuracy*, respectively.

3.2 Accuracy

We define two types of *accuracy* properties:

- *Strong accuracy*: Correct processes are never suspected.

$$\forall \sigma, \forall p \in up(\sigma), \forall q \in Q, \forall t : p \notin D_q(t, \sigma)$$

Since it is difficult (if not impossible) to achieve *strong accuracy*, we also define:

- *Weak accuracy*: Some correct process is never suspected.

$$\forall \sigma, \exists p \in up(\sigma), \forall q \in Q, \forall t : p \notin D_q(t, \sigma)$$

We define the following three classes of failure detectors:

[7] This definition is appropriate when the failure detector module associated with each process directly monitors a "local" subset of the entire system.

- $\mathcal{P}$, the set of *Perfect Failure Detectors* that satisfy the *strong completeness* and the *strong accuracy* properties.

- $\mathcal{S}$, the set of *Strong Failure Detectors* that satisfy the *strong completeness* and the *weak accuracy* properties.

- $\mathcal{W}$, the set of *Weak Failure Detectors* that satisfy the *weak completeness* and the *weak accuracy* properties.

Note that $\mathcal{P} \subset \mathcal{S} \subset \mathcal{W}$.

Even *weak failure detectors* guarantee that there is at least one correct process that is *never* suspected. Since this type of accuracy may be difficult to achieve, we consider weaker forms of accuracy in the next section.

3.3 Eventual Accuracy

We now consider failure detectors that may suspect *every* process at one time or another. Informally, we only require that some accuracy property is *eventually* satisfied. We consider the following two types of *eventual accuracy*.

- *Eventual strong accuracy*: There is a time after which correct processes are not suspected.

$$\forall \sigma, \exists t, \forall p \in up(\sigma), \forall q \in Q, \forall t' \geq t : p \notin D_q(t', \sigma)$$

- *Eventual weak accuracy*: There is a time after which some correct process is not suspected.

$$\forall \sigma, \exists t, \exists p \in up(\sigma), \forall q \in Q, \forall t' \geq t : p \notin D_q(t', \sigma)$$

For each one of the three types of failure detectors that we defined in Section 3.2, we can replace the *accuracy* requirement with the corresponding *eventual accuracy* requirement. This results in the following three classes of failure detectors:

- $\diamond\mathcal{P}$, the set of *Eventually Perfect Failure Detectors* that satisfy the *strong completeness* and the *eventual strong accuracy* properties.

- $\diamond\mathcal{S}$, the set of *Eventually Strong Failure Detectors* that satisfy the *strong completeness* and the *eventual weak accuracy* properties.

- $\diamond\mathcal{W}$, the set of *Eventually Weak Failure Detectors* that satisfy the *weak completeness* and the *eventual weak accuracy* properties.

Clearly, $\diamond\mathcal{P} \subset \diamond\mathcal{S} \subset \diamond\mathcal{W}$, and $\mathcal{P} \subset \diamond\mathcal{P}, \mathcal{S} \subset \diamond\mathcal{S}, \mathcal{W} \subset \diamond\mathcal{W}$ (Figure 1).

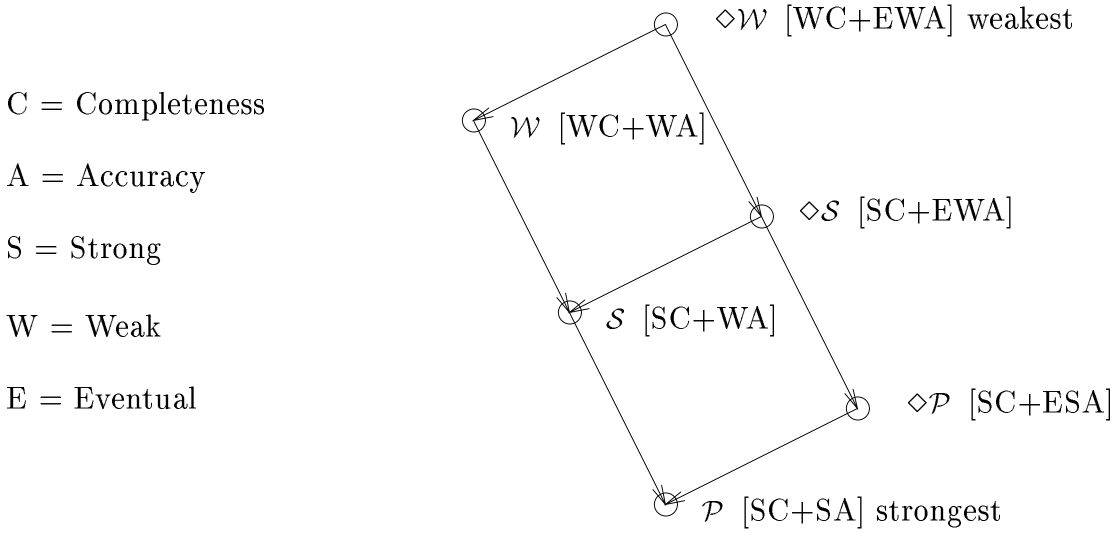

Figure 1: The lattice of failure detectors (by set containment).

3.4 Reductions

Informally, we say that the class of failure detectors $\mathcal{D}$ is *reducible* to the class of failure detectors $\mathcal{E}$, if for every failure detector $D \in \mathcal{D}$ there is an algorithm that can transform D into some failure detector $E \in \mathcal{E}$ (we denote this by $\mathcal{D} \sqsubseteq \mathcal{E}$). Note that if $\mathcal{D} \subseteq \mathcal{E}$ then $\mathcal{D} \sqsubseteq \mathcal{E}$. We say that $\mathcal{D}$ and $\mathcal{E}$ are *equivalent*, denoted $\mathcal{D} \cong \mathcal{E}$, if $\mathcal{D} \sqsubseteq \mathcal{E}$ and $\mathcal{E} \sqsubseteq \mathcal{D}$. The concept of reducibility will be formalized in the full version of this paper.

4 Spontaneous Reliable Broadcast

We now define a broadcast primitive that we repeatedly use in our algorithms. This primitive, *Spontaneous Reliable Broadcast*, can be used by any process to reliably broadcast a message at any time. Formally, Spontaneous Reliable Broadcast is defined in terms of two primitives, *broadcast*(m) and *deliver*(m) where m is a message drawn from a set of possible messages. When a process executes *broadcast*(m), we say that it *broadcasts* m, and when a process executes *deliver*(m), we say that it *delivers* m. The Spontaneous Reliable Broadcast problem is specified as follows[8]:

[8] For simplicity, we assume that each message is unique. In practice, this can be achieved by tagging the identity

{*The algorithm for a process p*}
procedure *broadcast*(m)
send (p, m) to all

when receive a message (q, m) for the first time:
deliver(m)
if $p \neq q$ **then** send (q, m) to all

Algorithm 1: Spontaneous Reliable Broadcast

- *Validity*: If a correct process broadcasts a message m, then all correct processes must eventually deliver m.
- *Agreement*: If a correct process delivers a message m, then all correct processes must eventually deliver m.
- *Integrity*: For any message m, a process delivers m at most once, and only if m was broadcast by some process.

Spontaneous Reliable Broadcast can be solved for asynchronous systems for several types of failures (note that the addition of a *termination* requirement would make this problem unsolvable in asynchronous systems). In particular, there is a simple algorithm that tolerates up to $n-1$ crash failures (see Algorithm 1). Essentially, when a process receives a message for the first time, it delivers the message and relays it.

5 The Spontaneous Consensus Problem

In the traditional formulation of the Consensus problem, all processes have initial values to propose, and correct processes must agree on some value which is somehow related to these initial values [Fis83]. We define the Consensus problem in terms of two primitives, *propose*(v) and *decide*(v), where v is a value drawn from a set of possible initial 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:

of the sender and a sequence number on each message.

- *Validity:* If a correct process decides v, then v was proposed by some process[9].

- *Agreement:* All correct processes that decide, decide the same value.

- *Termination:* Every correct process eventually decides some value.

This traditional formulation of Consensus assumes that all processes are aware that they must each propose and decide on a value. In this paper, we solve a stronger version of Consensus, called *Spontaneous Consensus*, in which processes may or may not wish to propose an initial value. Furthermore, if any process proposes an initial value, it may do so at any time. Thus, processes have no a priori knowledge of the other processes' intention to propose a value, or when they might do so. Spontaneous Consensus requires that *if* all correct processes eventually propose a value, *then* Termination, Validity, and Agreement are satisfied. If some correct process never proposes a value, then there are no requirements on whether processes decide, or on what values they decide. Clearly, any solution to Spontaneous Consensus is also a solution to Consensus.

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

6 Solving Spontaneous Consensus using a Weak Failure Detector

We first show how to convert any given Weak Failure Detector to a Strong Failure Detector. Then we show that given any Strong Failure Detector we can solve Spontaneous Consensus in asynchronous systems. Thus, given any Weak Failure Detector, Spontaneous Consensus is solvable in asynchronous systems.

6.1 From a Weak Failure Detector to a Strong Failure Detector

Algorithm 2 converts any Weak Failure Detector W in $\mathcal{W}$ to a Strong Failure Detector S in $\mathcal{S}$. Every process p reliably broadcasts the names of the processes that it suspects according to its failure detector module, W_p. Every process q that delivers such names, adds them to its list of suspects S_q.

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

{*The algorithm for a process* p}

$S_p \leftarrow \emptyset$ {p *uses* W_p *to construct* S_p}

cobegin

|| *Task 1:* **repeat forever**

  if $\exists q \in W_p - S_p$ **then** *broadcast*$(p, suspects, q)$

|| *Task 2:* **when** *deliver*$(r, suspects, s)$ for some r and s

  $S_p \leftarrow S_p \cup \{s\}$

coend

Algorithm 2: From a Weak Failure Detector $W \in \mathcal{W}$ to a Strong Failure Detector $S \in \mathcal{S}$.

Lemma 1.1: S satisfies *strong completeness.*

Proof: We first note that the sets S_p for all p are non-decreasing: Algorithm 2 never removes a process from S_p. Let s be any process that crashes. We must show that for *every* correct process q eventually $s \in S_q$.

By the *weak completeness* property of W, there is a time t after which $s \in W_p$ for *some* correct process p . We now show that eventually p adds s to S_p. For contradiction, suppose that s is never added to S_p. Then, after time t, s is permanently in $W_p - S_p$. Eventually *Task 1* of Algorithm 2 forces p to broadcast $(p, suspects, s)$. By the *validity* property of Spontaneous Reliable Broadcast, p delivers $(p, suspects, s)$ and adds s to S_p, a contradiction. So, p adds s to S_p.

To add s to S_p, p must execute *Task 2*: p must deliver $(r, suspects, s)$ for some r. By the *agreement* property of Spontaneous Reliable Broadcast, every correct process q eventually delivers $(r, suspects, s)$ and also adds s to S_q. □

Lemma 1.2: S satisfies *weak accuracy.*

Proof: By the *weak accuracy* property of W, there is a correct process q that is never suspected by any process, i.e., $q \notin W_p$ for all p. Thus, no process ever broadcasts a message of type $(-, suspects, q)$. So no process ever delivers a message of type $(-, suspects, q)$ and for all p,

$q \notin S_p$. □

Theorem 1: Algorithm 2 reduces $\mathcal{W}$ to $\mathcal{S}$.
Proof: Follows from Lemmas 1.1 and 1.2. □

6.2 The Spontaneous Consensus Algorithm

Given any Strong Failure Detector S, Algorithm 3 solves the Spontaneous Consensus problem in asynchronous systems. Algorithm 3 runs through 3 phases. In Phase 1, processes execute n asynchronous rounds (r_p denotes the current round number of process p) during which they broadcast and relay their initial values. At the end of each asynchronous round, each process p waits until, for each process q, either p receives a message from q or q is suspected to have crashed by p's failure detector module S_p. By the end of Phase 2, correct processes agree on a vector based on the initial values of all processes. We will show that this vector contains the initial value of at least one process. In Phase 3, correct processes decide the first non-trivial component of this vector.

Note that even though we assume that at most f processes may crash[10], Phase 1 of the algorithm consists of n rounds, rather than the usual $f+1$ rounds of traditional Consensus algorithms (for synchronous systems). Intuitively, this is because even a *correct* process p may be suspected to have crashed by other processes. In this case, p's messages may be ignored, and p appears to commit "send-omission" failures. Thus, up to $n-1$ processes may appear to commit failures (rather than f).

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

Assume that *all* correct processes eventually propose a value. We have to show that *validity*, *agreement* and *termination* hold.

Lemma 2.1: For all p and q, $V_p[q]$ is either v_q or $\perp$.
Proof: Obvious from Algorithm 3. □

Lemma 2.2: Every correct processes eventually decides some value.
Proof: [*sketch*] The only way a correct process p can be prevented from deciding is by blocking forever at one of the two **wait** statements. This can happen only if p is waiting forever for a message from a process q and q never joins S_p. There are two cases to consider:

[10] The maximum number of faulty processes is usually denoted by t rather than f, the notation adopted here. We reserve t to denote real-time.

$\{$*The algorithm for a process* $p\}$

procedure *propose*(v_p) $\{p$ *proposes its initial value* $v_p\}$

$V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$ $\{p$*'s estimate of the initial values*$\}$

$V_p[p] \leftarrow v_p$

$\Delta_p \leftarrow V_p$

Phase 1: $\{$*asynchronous rounds* r_p, $1 \le r_p \le n\}$

for $r_p \leftarrow 1$ to n

send (r_p, Δ_p, p) to all

wait until $[\forall q:$ received (r_p, Δ_q, q) **or** $q \in S_p]$ $\{$*Query the failure detector*$\}$

$msgs_p(r_p) \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$

$\Delta_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$

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** $q \in S_p]$ $\{$*Query the failure detector*$\}$

$lastmsgs_p \leftarrow \{V_q \mid \text{received } V_q\}$

for $k \leftarrow 1$ to n

if $\exists V_q \in lastmsgs_p$ with $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

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

Algorithm 3: Solving Spontaneous Consensus using a Strong Failure Detector $S \in \mathcal{S}$.

1. If q crashes then the *strong completeness* property of the failure detector S ensures that eventually $q \in S_p$.

2. If q does not crash then 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. $\square$

From the *weak accuracy* property of S, there is a correct process c that is never suspected (i.e., for all processes p, $c \notin S_p$). Let P_2 be the set of processes that reach Phase 2 (i.e., they completed all n rounds of Phase 1).

Lemma 2.3: In every round r, $1 \leq r \leq n$, all processes p in P_2 receive (r, Δ_c, c) from process c, i.e., (r, Δ_c, c) is in $msgs_p(r)$.
Proof: Since $p \in P_2$, p completes all n rounds. At each round r, since $c \notin S_p$, p waits for and receives the message (r, Δ_c, c) from c. $\square$

Lemma 2.4: Suppose $V_c[q] = v_q$ for some process q at the end of Phase 1. Then for all p in P_2, $V_p[q] = v_q$ at the end of Phase 1.
Proof: Let r be the first round in which c received v_q (if $c = q$, we define r to be 0). Since any process relays v_q (in the vector Δ) at most once, it is easy to see that $r \leq n-1$. Thus, in round $r+1 \leq n$, c relays v_q by sending the message $(r+1, \Delta_c, c)$ with $\Delta_c[q] = v_q$ to all. From Lemma 2.3, all $p \in P_2$ receive $(r+1, \Delta_c, c)$ and thus $V_p[q] = v_q$ by the end of round $r+1$. $\square$

Lemma 2.5: All processes that decide, decide the same value[11].
Proof: Each process p that decides, decides the first non-trivial value of its vector V_p. We now show that at the end of Phase 2, $V_p = V_c$ and the lemma follows.

Let q be any process. There are two cases to consider:

1. $V_c[q] = v_q$ at the end of Phase 1. From Lemma 2.4, for all processes $p' \in P_2$ (including p and c), $V_{p'}[q] = v_q$ at the end of Phase 1. Thus all the vectors V that are sent in Phase 2, are such that $V[q] = v_q$. Thus both $V_p[q]$ and $V_c[q]$ remain equal to v_q throughout Phase 2.

2. $V_c[q] = \perp$ at the end of Phase 1. Since $c \notin S_p$, p waits for and receives V_c in Phase 2. Since $V_c[q] = \perp$, p sets $V_p[q] \leftarrow \perp$ at the end of Phase 2. $\square$

[11] This property is stronger than the *agreement* requirement of Spontaneous Consensus which applies only to *correct* processes.

Theorem 2: Given any Strong Failure Detector, Algorithm 3 solves the Spontaneous Consensus problem in asynchronous systems with $f < n$.

Proof: By Lemma 2.2 and 2.5, Algorithm 3 satisfies *termination* and *agreement*, respectively. At initialization, $V_c[c]$ is set to v_c. Thus at the end of Phase 1, $V_c[c] = v_c$. From Case 1 of the proof of Lemma 2.5 (with $q = c$), every correct process p has $V_p[c] = v_c$ at the end of Phase 2, and decides a non-$\perp$ value in Phase 3. By Lemma 2.1, this must be the initial value v_q of some process q. Thus Algorithm 3 satisfies *validity*. $\square$

From Theorems 1 and 2 we have:

Corollary 2.1: Given any Weak Failure Detector, Spontaneous Consensus is solvable in asynchronous systems with $f < n$.

7 Solving Spontaneous Consensus using an Eventually Weak Failure Detector

Our previous solution to the Spontaneous Consensus problem relied on failure detectors with *weak accuracy*: some correct process is never suspected. We now solve the Spontaneous Consensus problem given a failure detector with *eventual weak accuracy.* With this failure detector *all* processes may be erroneously added to the lists of suspects at one time or another. However, there is a correct process and a time after which that process is not suspected to have crashed. Note that at any given time t, processes cannot use this failure detector to determine whether any specific process is correct, or whether some correct process will never be suspected after time t.

We first show how to convert any Eventually Weak Failure Detector to an Eventually Strong Failure Detector. Then we show that given any Eventually Strong Failure Detector we can solve Spontaneous Consensus in asynchronous systems with $f < \frac{n}{2}$ (the requirement that $f < \frac{n}{2}$ is shown to be necessary in Appendix A). Thus, given any Eventually Weak Failure Detector, Spontaneous Consensus is solvable in asynchronous systems.

7.1 From an Eventually Weak Failure Detector to an Eventually Strong Failure Detector

Algorithm 4 converts any Eventually Weak Failure Detector W in $\diamond\mathcal{W}$ to an Eventually Strong Failure Detector S in $\diamond\mathcal{S}$. Informally, the conversion is as follows. Every process p periodically broadcasts the names of the processes that it suspects according to its failure detector

$\{$*The algorithm for a process* $p\}$

$S_p \leftarrow \emptyset$ $\{$*Failure detector module constructed by* $p\}$

for all $q, r \in Q$

  $suspected_p(r, q) \leftarrow 0$ $\{$*Number of times* q *suspected* r *according to* $p\}$

  $refuted_p(r, q) \leftarrow 0$ $\{$*Number of times* r *refuted* q *according to* $p\}$

cobegin

$||$ *Task 1:* **repeat forever**

  for all processes r

    if $r \in W_p$ **and** $refuted_p(r, p) \geq suspected_p(r, p)$ **then**

      broadcast$(p, suspects, r, refuted_p(r, p) + 1)$

$||$ *Task 2:* **when** *deliver*$(q, suspects, r, j)$

    $suspected_p(r, q) \leftarrow \max(suspected_p(r, q), j)$

    if $p = r$ **then** *broadcast*$(p, refutes, q, j)$

$||$ *Task 3:* **when** *deliver*$(r, refutes, q, j)$

    $refuted_p(r, q) \leftarrow \max(refuted_p(r, q), j)$

$||$ *Task 4:* **repeat forever**

    for all processes r

      if $\exists q : suspected_p(r, q) > refuted_p(r, q)$ **then** $S_p \leftarrow S_p \cup \{r\}$

      else $S_p \leftarrow S_p - \{r\}$

coend

Algorithm 4: From an Eventually Weak Failure Detector $W \in \diamond\mathcal{W}$ to an Eventually Strong Failure Detector $S \in \diamond\mathcal{S}$.

module, W_p. Every process q that delivers such names, adds them to its list of suspects S_q. These broadcasts and the *weak completeness* property of W result in the *strong completeness* property of the constructed S. If a process p discovers that it was erroneously suspected by a process, p broadcasts an appropriate *refutation*, i.e., a proof that the suspicion was a mistake. Upon delivering p's refutation, a process q removes p from its list of suspects S_q. However, p can be suspected again and added to S_q a second time. Following a second refutation from p, p is again removed from S_q. Thus a possibly infinite sequence of suspicions followed by corresponding refutations may occur—resulting in p being repeatedly added and then removed from S_q. However, by the *eventual weak accuracy* property of W, there is a time after which *some* correct process is not suspected. Together with the refutation mechanism, this ensures the *eventual weak accuracy* property of the constructed S.

Lemma 3.1: S satisfies *eventual weak accuracy.*

Proof: Since W satisfies *eventual weak accuracy*, there is a correct process r and a time t after which no process suspects r. After time t, no process broadcasts a message of type $(-, suspects, r, -)$. We will show that for every correct process p, there is a time t_p after which, for all processes q, $refuted_p(r,q) \geq suspected_p(r,q)$ is always true. Thus there is a time after which Task 4 of Algorithm 4 forces p to permanently remove r from S_p.

Consider any process q. If p never delivers a message of type $(q, suspects, r, -)$ then $refuted_p(r,q) \geq suspected_p(r,q)$ is always true (since $suspected_p(r,q)$ always equals 0). Now assume that p does deliver at least one message of type $(q, suspects, r, -)$. Let k denote the maximum value of j for which $(q, suspects, r, j)$ is delivered to p. Thus there is a time after which $suspected_p(r,q) = k$. By the *agreement* property of Spontaneous Reliable Broadcast, r eventually delivers $(q, suspects, r, k)$. When it does so, Task 2 of Algorithm 4 forces r to broadcast $(r, refutes, q, k)$. Since r is a correct process, the *validity* property of Spontaneous Reliable Broadcast ensures that p eventually delivers $(r, refutes, q, k)$ in Task 3. After it does so, $refuted_p(r,q) \geq k$. We conclude that thereafter $refuted_p(r,q) \geq suspected_p(r,q)$ is always true. □

Lemma 3.2: S satisfies *strong completeness.*

Proof: Consider a process r that crashes. Since W satisfies *weak completeness*, there is a correct process q and a time t after which r is permanently in W_q. We will show that for every correct process p, there is a time after which $suspected_p(r,q) > refuted_p(r,q)$ is always true. Thus there is a time after which Task 4 of Algorithm 4 forces p to permanently include r in S_p.

Since r crashes, it can only broadcast a finite number of messages of type $(r, refutes, q, -)$. Let k be the maximum value of j for which $(r, refutes, q, j)$ is delivered to p (the case when

p never delivers such a message is similar and we omit it). Thus there is a time after which $refuted_p(r, q) = k$. By the *agreement* property of Spontaneous Reliable Broadcast, q eventually delivers $(r, refutes, q, k)$ and q sets $refuted_q(r, q) \geq k$. Since after time t, r is permanently in W_q, we claim that Task 1 of Algorithm 4 eventually forces q to broadcast $(q, suspects, r, k')$ with $k' > k$.

Suppose, by contradiction, q never broadcasts $(q, suspects, r, k')$ with $k' > k$. Then $suspected_q(r, q) \leq k$. Since q eventually sets $refuted_q(r, q) \geq k$, there is a time after which $refuted_q(r, q) \geq suspected_q(r, q)$. Since after time t, r is permanently in W_q, Task 1 of Algorithm 4 eventually forces q to broadcast $(q, suspects, r, k')$ with $k' > k$—a contradiction that proves our claim.

Since q is a correct process, the *validity* property of Spontaneous Reliable Broadcast ensures that p eventually delivers $(q, suspects, r, k')$ in Task 3. After it does so, $suspected_p(r, q) \geq k'$. Since there is a time after which $refuted_p(r, q) = k$, there is a time after which $suspected_p(r, q) > refuted_p(r, q)$ is always true. □

Theorem 3: Algorithm 4 reduces $\diamond\mathcal{W}$ to $\diamond\mathcal{S}$.
Proof: From Lemma 3.1 and Lemma 3.2. □

7.2 The Spontaneous Consensus Algorithm

Given any Eventually Strong Failure Detector S, Algorithm 5 solves the Spontaneous Consensus problem in asynchronous systems with $f \leq \frac{n}{2}$. Algorithm 5 uses the *rotating coordinator* paradigm [Rei82, CM84, DLS88, CT90]. Computation proceeds in asynchronous "rounds". We assume that all processes have a priori knowledge that during round r, the coordinator is process $c = (r \bmod n) + 1$. All messages are either to or from the "current" coordinator. Every time a process becomes a coordinator, it tries to determine a consistent decision value. If the current coordinator is correct *and* is not suspected by any surviving process, then it will succeed, and it will broadcast this decision value using a Spontaneous Reliable Broadcast.

Algorithm 5 goes through three asynchronous epochs, each of which may span several asynchronous rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked*: no other decision value is possible. In the third epoch, processes decide the locked value.

Each round of Algorithm 5 is divided into four asynchronous phases. In Phase 1, every process sends its current estimate of the decision value timestamped with the round number in which it adopted this estimate, to the current coordinator, c. In Phase 2, c gathers $n - f$ such estimates, selects one with the largest timestamp, and sends it to all the processes as their new estimate,

{*The algorithm for a process* p}

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

  {*Rotate through coordinators until decision is reached*}

  while $state_p = undecided$
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ {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* $n - f$ *estimates and proposes a new estimate*}
      if $p = c_p$ **then**
        wait until [for $n - f$ processes q : received $(q, r_p, estimate_q, ts_q)$ from q]
        $msgs_p(r_p) \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ 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** $c_p \in S_p$]{*Query the failure detector*}
      if [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then** {p *received* $estimate_{c_p}$ *from* c_p}
        $estimate_p \leftarrow estimate_{c_p}$
        $ts_p \leftarrow r_p$
        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: {*The current coordinator waits for* $n - f$ *replies. If these replies indicate that* $n - f$ *processes adopted its estimate, the coordinator sends a request to decide.*}
      if $p = c_p$ **then**
        wait until [for $n - f$ processes q : received (q, r_p, ack) **or** $(q, r_p, nack)$]
        if [for $n - f$ processes q : received (q, r_p, ack)] **then** $broadcast(p, r_p, estimate_p, decide)$

{*When* p *receives a decide message, it decides*}

when $deliver(q, r_q, estimate_q, decide)$
  $decide(estimate_q)$
  $state_p \leftarrow decided$

Algorithm 5: Solving Spontaneous Consensus using an Eventually Strong Failure Detector $S \in \diamond\mathcal{S}$.

$estimate_c$. 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.

2. Upon consulting the failure detector S, p *suspects* that c crashed, and sends a *nack* to c.

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

The proof that Algorithm 5 solves Spontaneous Consensus is as follows. Assume that *all* correct processes eventually propose a value. We have to show that *validity*, *agreement* and *termination* hold.

Lemma 4.1: All processes that decide, decide the same value[12].

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

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

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

[12] This property is stronger than the *agreement* requirement of Spontaneous Consensus which applies only to *correct* processes.

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 Algorithm 5, 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 delivers $(q, r_q, estimate_q, decide)$, and thus decides $estimate_q$. Process q must have broadcast $(q, r_q, estimate_q, decide)$ in Phase 4 of round r_q. From Algorithm 5, q must have received $n - f$ messages of type $(-, r_q, ack)$ in Phase 4 of round r_q. By the definition of r, $r \leq r_q$. From the above claim, $estimate_q = estimate_c$. □

Lemma 4.2: Every correct process eventually decides some value.
Proof: There are two possible cases:

1. Some correct process decides. It must have delivered some message of type $(-, -, -, decide)$. By the *agreement* property of Spontaneous Reliable Broadcast, all correct processes eventually 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 type $(-, r, estimate, -)$ to the current coordinator $c = (r \bmod n) + 1$. Therefore at least $n - f$ such messages are sent to c. There are two cases to consider:

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

   (b) c crashes.

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

   From the *eventual weak accuracy* property of the failure detector S, there is a correct process

q and a time t such that no process suspects q after t. From this and the above claim, there must be a round r such that:

(a) All correct processes reach round r after time t (when no process suspects q).

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

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

Theorem 4: Given any Eventually Strong Failure Detector, Algorithm 5 solves the Spontaneous Consensus problem in asynchronous systems with $f < \frac{n}{2}$.

Proof: By Lemma 4.1 and 4.2, Algorithm 5 satisfies *agreement* and *termination*, respectively. From the algorithm, it is clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision values that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, *validity* is satisfied. $\square$

From Theorems 3 and 4 we have:

Corollary 4.1: Given any Eventually Weak Failure Detector, Spontaneous Consensus is solvable in asynchronous systems with $f < \frac{n}{2}$.

8 Using Spontaneous Consensus to Solve Atomic Broadcast

Informally, Atomic Broadcast requires that all correct processes deliver the same messages in the same order. Formally, *Atomic Broadcast* is a Spontaneous Reliable Broadcast that satisfies:

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

Total order and *agreement* ensure that all correct processes deliver the same *sequence* of messages. Atomic Broadcast is perhaps the most powerful communication paradigm for fault-tolerant

{*The algorithm for a process p*}

Initialisation:

$ab\_broadcast_p \leftarrow \emptyset$

$ab\_delivered_p \leftarrow \emptyset$

$k \leftarrow 0$

procedure *atomic-broadcast*(m) {*Task 1*}

broadcast(m)

when *deliver*(m) {*Task 2*}

$ab\_broadcast_p \leftarrow ab\_broadcast_p \cup \{m\}$

when $(ab\_broadcast_p - ab\_delivered_p) \neq \emptyset$ {*Task 3*}

$k \leftarrow k + 1$

$ab\_undelivered_p \leftarrow ab\_broadcast_p - ab\_delivered_p$

$propose(k, ab\_undelivered_p)$

wait until $decide(k, msg\_set)$

$batch_p(k) \leftarrow msg\_set - ab\_delivered_p$

atomically deliver all messages in $batch_p(k)$ in some deterministic order

$ab\_delivered_p \leftarrow ab\_delivered_p \cup batch_p(k)$

Algorithm 6: Solving Atomic Broadcast using Spontaneous Consensus.

distributed computing [CM84,CASD85,Sch86,BJ87,PGM89,BGT90,GSTC90]. Unfortunately, Atomic Broadcast cannot be solved in asynchronous systems that are subject to even a single crash failure (in fact, one can easily reduce Spontaneous Consensus to Atomic Broadcast). In this section, we show how to solve Atomic Broadcast in asynchronous systems with failure detectors.

We show that given any Spontaneous Consensus algorithm, we can solve Atomic Broadcast. In Sections 6 and 7, we described Spontaneous Consensus algorithms for asynchronous systems with two types of failure detectors. Thus Atomic Broadcast can be solved in asynchronous systems with these failure detectors.

Our reduction from Atomic Broadcast to Spontaneous Consensus applies to asynchronous systems, but does *not* require the assumption of a failure detector. This shows that Spontaneous Consensus and Atomic Broadcast are *equivalent* in asynchronous systems with crash failures. In fact, in [CHT91] we use this reduction to derive *randomized* solutions to Atomic Broadcast from randomized solutions to Spontaneous Consensus [Ben83,CD89].

Algorithm 6 shows how to implement Atomic Broadcast in asynchronous systems by using repeated (possibly concurrent, but completely *independent*) executions of Spontaneous Consensus. Processes disambiguate between these independent executions by tagging a counter k to all the messages pertaining to the k^{th} execution of Spontaneous Consensus. Tagging these counters constitute a minor modification to our Spontaneous Consensus algorithms. Informally, the k^{th} execution of Spontaneous Consensus is used to agree on the k^{th} batch of messages to be atomically delivered. The propose and decide primitives corresponding to the k^{th} execution of Spontaneous Consensus are denoted by $propose(k, -)$ and $decide(k, -)$.

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

When a process intends to atomically broadcast a message m, it broadcasts m (using Spontaneous Reliable Broadcast, in Task 1). When a process p delivers m, it adds m to the set $ab\_broadcast_p$ (Task 2). Thus $ab\_broadcast_p$ contains all the messages submitted for Atomic Broadcast that p is currently aware of. When p atomically delivers a message m, it adds m to the set $ab\_delivered_p$ (in Task 3). Thus $ab\_broadcast_p - ab\_delivered_p$ is the set of messages that were submitted for Atomic Broadcast but not yet atomically delivered by p. This set is denoted by $ab\_undelivered_p$. When $ab\_undelivered_p$ is not empty, p proposes $ab\_undelivered_p$ as the next batch of messages to be atomically delivered. $batch_p(k)$ denotes the k^{th} batch of messages that p atomically delivers: it is msg_set_p, the set of messages agreed upon by the k^{th} execution of Spontaneous Consensus, minus $ab\_delivered_p$, those messages that p has already atomically delivered[13].

Lemma 5.1: For any two correct processes p and q, and any message m, if $m \in ab\_broadcast_p$ then eventually $m \in ab\_broadcast_q$.

Proof: If $m \in ab\_broadcast_p$ then p delivered m (in Task 2). Since p is correct, by the agreement property of Spontaneous Reliable Broadcast q eventually delivers m, and includes m in $ab\_broadcast_q$. $\square$

Let t_p^k denote the time at which process p completes the execution of the statement $ab\_delivered_p \leftarrow ab\_delivered_p \cup batch_p(k)$ (at the end of p's k^{th} execution of Task 3). If p never executes this step, $t_p^k = \infty$.

Lemma 5.2: For any two correct processes p and q, and all $k \geq 1$:

1. If p executes $propose(k, ab\_undelivered_p)$, then q eventually executes $propose(k, ab\_undelivered_q)$.
2. The sequence of messages that p atomically delivers by time t_p^k is the same as the sequence of messages that q atomically delivers by time t_q^k.

Proof: The proof is by simultaneous induction on 1 and 2. For $k = 1$, we show that if p executes $propose(1, ab\_undelivered_p)$, then q also executes $propose(1, ab\_undelivered_q)$. When p executes $propose(1, ab\_undelivered_p)$ in Task 3, the set $ab\_undelivered_p$ contains at least one message m that is in $ab\_broadcast_p$. By Lemma 5.1, m is eventually in $ab\_broadcast_q$. Since $ab\_delivered_q$ remains empty till the first execution of Spontaneous Consensus by q, eventually $ab\_broadcast_q - ab\_delivered_q \neq \emptyset$. Thus, eventually q executes Task 3 and $propose(1, ab\_undelivered_q)$. By the *agreement* and *termination* properties of Spontaneous Consensus, both p and q decide the same set of messages, msg_set. Since $ab\_delivered_p$ and $ab\_delivered_q$ are both still empty, $batch_p(1) = batch_q(1) = msg\_set$, and hence p and q atomically deliver messages in msg_set in the same order (by time t_p^1 and t_q^1, respectively).

Now assume that the lemma holds for all k, $1 \leq k < l$. We first show that part (1) of the lemma is true for l. For contradiction, assume that p executes $propose(l, ab\_undelivered_p)$ and q never executes $propose(l, ab\_undelivered_q)$. When p executes $propose(l, ab\_undelivered_p)$, $ab\_undelivered_p$ must contain at least one message, m. This execution occurs after time t_p^{l-1}.

[13] It is possible for a process to atomically deliver a message m before it delivers it via Spontaneous Reliable Broadcast. This occurs if m is proposed by another process, and agreed upon by Consensus, before p delivers m.

So some time after t_p^{l-1}, m is in $ab\_broadcast_p$ but not in $ab\_delivered_p$. By Lemma 5.1, m is eventually in $ab\_broadcast_q$. We now show that m is never in $ab\_delivered_q$.

By the induction hypothesis, the sequence of messages that q atomically delivered by time t_q^{l-1} is the same as the sequence of messages that p atomically delivered by time t_p^{l-1}. Since m is not in $ab\_delivered_p$ at time t_p^{l-1}, m is not in $ab\_delivered_q$ at time t_q^{l-1}. Since q does not execute Spontaneous Consensus after time t_q^{l-1}, m is never in $ab\_delivered_q$.

Thus there is a time after which m is permanently in $ab\_broadcast_q - ab\_delivered_q$. So eventually q executes Task 3 and $propose(l, ab\_undelivered_q)$—a contradiction. This concludes the proof of part (1) of this lemma.

Now we show that part (2) of this lemma holds for $k = l$. By the induction hypothesis, p and q atomically delivered the same sequence of messages by time t_p^{l-1} and t_q^{l-1}, respectively. Thus we only have to show that p and q atomically deliver the same sequence of messages during the intervals $[t_p^{l-1}, t_p^l]$ and $[t_q^{l-1}, t_q^l]$, respectively.

By the first part of the lemma, either both p and q execute $propose(l, -)$ or neither do so. If neither p nor q execute $propose(l, -)$, then they do not atomically deliver any messages after t_p^{l-1} and t_q^{l-1}, respectively. If both execute $propose(l, -)$, then, from the *agreement* and *termination* properties of Spontaneous Consensus, both p and q decide the same set of messages msg_set. By the induction hypothesis, the value of $ab\_delivered_p$ when p executes $batch_p(l) \leftarrow msg\_set - ab\_delivered_p$ is the same as the value of $ab\_delivered_q$ when q executes $batch_q(l) \leftarrow msg\_set - ab\_delivered_q$. Thus $batch_p(l) = batch_q(l)$. So, p and q atomically deliver the same messages in the same deterministic order, during the intervals $[t_p^{l-1}, t_p^l]$ and $[t_q^{l-1}, t_q^l]$, respectively. This concludes the proof of part (2) of the lemma. □

Lemma 5.3: If a correct process atomically broadcasts a message m, then every correct process eventually atomically delivers m.

Proof: Let p be any correct process that atomically broadcasts a message m. By Lemma 5.2, if any correct process atomically delivers m, then all correct processes do so. Suppose by contradiction, that correct processes never atomically deliver m. By Task 1 of Algorithm 6, p broadcasts m. By the *validity* property of Spontaneous Reliable Broadcast, every correct process q eventually delivers m, and includes m in $ab\_broadcast_q$. Since correct processes never atomically deliver m, m is never included in $ab\_delivered_q$. Thus for every correct process q, there is a time after which m is permanently in $ab\_broadcast_q - ab\_delivered_q$. So every correct process q repeatedly proposes $ab\_undelivered_q$, with m in $ab\_undelivered_q$.

Let t be the time after which no process crashes. There is a time $t' > t$ such that after t':

1. Only correct processes execute Spontaneous Consensus.

2. For every correct process q, m is in $ab\_undelivered_q$.

Thus by the *validity* property of Spontaneous Consensus, all correct processes eventually decide some $ab\_undelivered_q$ that contains m. Thus all correct processes atomically deliver m—a contradiction that concludes the proof. □

Lemma 5.4: For any message m, a process atomically delivers m at most once, and only if m was atomically broadcast by some process.

Proof: It is easy to see that a process atomically delivers m, only if m was indeed atomically broadcast by some process. Suppose that p atomically delivers m. Let l be the smallest value of k for which m is an element of $batch_p(k)$. Thus p atomically delivers m by time t_p^l. Thereafter m is in $ab\_delivered_p$, and p cannot include m in any $batch_p(k)$ with $k > l$. So p does not atomically deliver m again, and this satisfies *integrity*. □

Theorem 5: Algorithm 6 reduces Atomic Broadcast to Spontaneous Consensus in an asynchronous system.

Proof: *Agreement* and *order* follow from Lemma 5.2. *Validity* and *integrity* follow from Lemma 5.3 and Lemma 5.4, respectively. □

Corollary 5.1: Given any Weak Failure Detector, Atomic Broadcast is solvable in asynchronous systems with $f < n$.

Corollary 5.2: Given any Eventually Weak Failure Detector, Atomic Broadcast is solvable in asynchronous system with $f < \frac{n}{2}$.

9 Related Work

The impossibility of Consensus in the asynchronous model of computation was shown in [FLP85]. Dolev et. al. extended this result by determining the possibility/impossibility of Consensus for several models of computation [DDS87]. These models range from the completely synchronous model (synchronous processes and communication) to the completely asynchronous model (asynchronous processes and communication).

In [DLS88], Dwork et. al. define two more models that lie between the completely synchronous and completely asynchronous cases, with several types of failures, and describe corresponding Consensus algorithms (one of which is similar to Algorithm 5). The first model assumes that there are bounds on relative process speeds and on message transmission times, but these bounds are not known. The second model assumes that these bounds are known, but they hold only after some unknown time. However, [DLS88] did not consider a weaker model of partial synchrony, in which the bounds are not known *and* they hold only after some unknown time. For each one of these three models (with crash failures), it is easy to implement an Eventually Perfect Failure Detector, and thus solve Consensus using Algorithm 5[14] (Appendix B has more details).

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 atomically deliver the same messages as other correct processes, and in the same order. Furthermore, it can continue to atomically broadcast messages. In [RB91], Ricciardi and Birman take an alternative approach which is based on the assumption that failure detectors rarely make mistakes. In those infrequent cases in which a correct process p is falsely suspected to have crashed, p is effectively removed from the system (via a *group membership* protocol). For example, with Atomic Broadcast, p is forced to stop atomically broadcasting and atomically delivering messages. Essentially, this approach forces the system to conform with the view of the failure detector. This is reasonable since, in most systems, failure detectors can be tuned to make few mistakes.

Schlichting and Schneider proposed a model of failure called *fail-stop* that is more "benign" than crash failures [SS83,Sch84]. A fail-stop processor can only fail by prematurely halting, but it is assumed that "Any processor can detect when any other processor has failed ...". (It is also assumed that each processor has stable storage "...that is unaffected by any failure and is accessible to every processor.") Thus, a system with fail-stop processes is similar to a system where processes can crash and have access to a Perfect Failure Detector.

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

Acknowledgements

We are deeply grateful to Vassos Hadzilacos for his help in revising this paper. We would also like to thank Özalp Babaoğlu, Ken Birman, Navin Budhiraja, Prasad Jayanti, Dexter Kozen, Keith Marzullo, Gil Neiger, Anil Nerode, Aleta Ricciardi, Fred Schneider and the distributed systems group at Cornell for their comments and criticisms.

[14] Actually, for Algorithm 5 to work, a weaker failure detector—the Eventually Strong Failure Detector is sufficient.

References

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

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

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

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

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

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

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

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

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

[CHT91] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. On the equivalence of consensus and atomic broadcast, personal communication. 1991.

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

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

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

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

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

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

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

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

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

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

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

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

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

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sch86] Fred B. Schneider. The state machine approach: a tutorial. Technical Report 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

Appendix A: A Hierarchy of Failure Detectors and Lower Bounds

In this section, we introduce a hierarchy of failure detectors that will be used to unify our results and to state a lower bound. This hierarchy consists of a continuum of failure detectors ordered by the number of "mistakes" that each one can make.

Mistakes and Repentance

Informally, a mistake is the *erroneous* addition of a process q that has not crashed to the list of processes that are suspected to have crashed. Formally, a failure detector D makes a *mistake* at time t at process p about process q in run σ if:

$$[q \in up(t, \sigma), q \in D_p(t, \sigma)] \text{ and } [\exists t' < t, \forall t'', t' \leq t'' < t : q \notin D_p(t'', \sigma)]$$

Such a mistake is denoted by the tuple $\langle p, q, t, \sigma, D\rangle$. The set of mistakes made by D in run σ is denoted by $M(\sigma, D)$.

Note that each erroneous *addition* of q into D_p is counted as a *single* mistake at p. The continuous *retention* of q into that set 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 D_p. In practice, mistakes occur due to *premature* time-outs.

We define the following four types of accuracy for a failure detector D:

- *Strongly $k-$mistaken:* D makes at most k mistakes. Formally:

  $$\forall \sigma : |M(\sigma, D)| \leq k$$

- *Weakly $k-$mistaken:* There is at least one correct process p such that D makes at most k mistakes about p. Formally:

  $$\forall \sigma, \exists p \in up(\sigma) : |\{\langle q, p, t, \sigma, D\rangle : \langle q, p, t, \sigma, D\rangle \in M(\sigma, D)\}| \leq k$$

- *Strongly finitely mistaken:* D makes a finite number of mistakes. Formally:

  $$\forall \sigma : |M(\sigma, D)| \text{ is finite.}$$

  Clearly there is a time t after which D stops making mistakes.

- *Weakly finitely mistaken:* There is at least one correct process p such that D makes a finite number of mistakes about p. Formally:

  $$\forall \sigma, \exists p \in up(\sigma) : |\{\langle q, p, t, \sigma, D\rangle : \langle q, p, t, \sigma, D\rangle \in M(\sigma, D)\}| \text{ is finite.}$$

  In this case, there is a time t after which D stops making mistakes about p.

For most values of k, the properties mentioned above are not powerful enough to be useful. For example, suppose every process always suspects every other process. In this case, the failure detector makes at most n^2 mistakes, and 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 if this mistake is "obvious". Fortunately, we can impose a natural requirement to circumvent this problem. Consider the following scenario. The failure detector at process p *erroneously* adds q to 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, and p knows that its failure detector made a mistake about q. It is reasonable to require that, given such irrefutable evidence of a mistake, the failure detector at p takes the corrective action of removing q from D_p. In general we can require the following property:

- *Repentance:* If a process p *knows* that its failure detector made a mistake at time t about process q, then at some time after t, q is removed from D_p.

$$
\begin{aligned}
&\forall \sigma, t, \forall p, q \in Q : \\
&\quad \{[q \in D_p(t, \sigma)] \text{ and } [\exists t' > t : (\sigma, t') \models K_p(q \in up(t, \sigma))]\} \\
&\quad \Rightarrow [\exists t'' > t : q \notin D_p(t'', \sigma)]
\end{aligned}
$$

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

The Hierarchy

We define a hierarchy of failure detectors that satisfy the *weak completeness* and the *repentance* properties. They differ only on the type of accuracy that they satisfy:

- Failure detectors in $\mathcal{SF}(k)$ are *strongly $k-$mistaken.*

- Failure detectors in $\mathcal{SF}$ are *strongly finitely mistaken.*

- Failure detectors in $\mathcal{WF}(k)$ are *weakly $k-$mistaken.*

- Failure detectors in $\mathcal{WF}$ are *weakly finitely mistaken.*

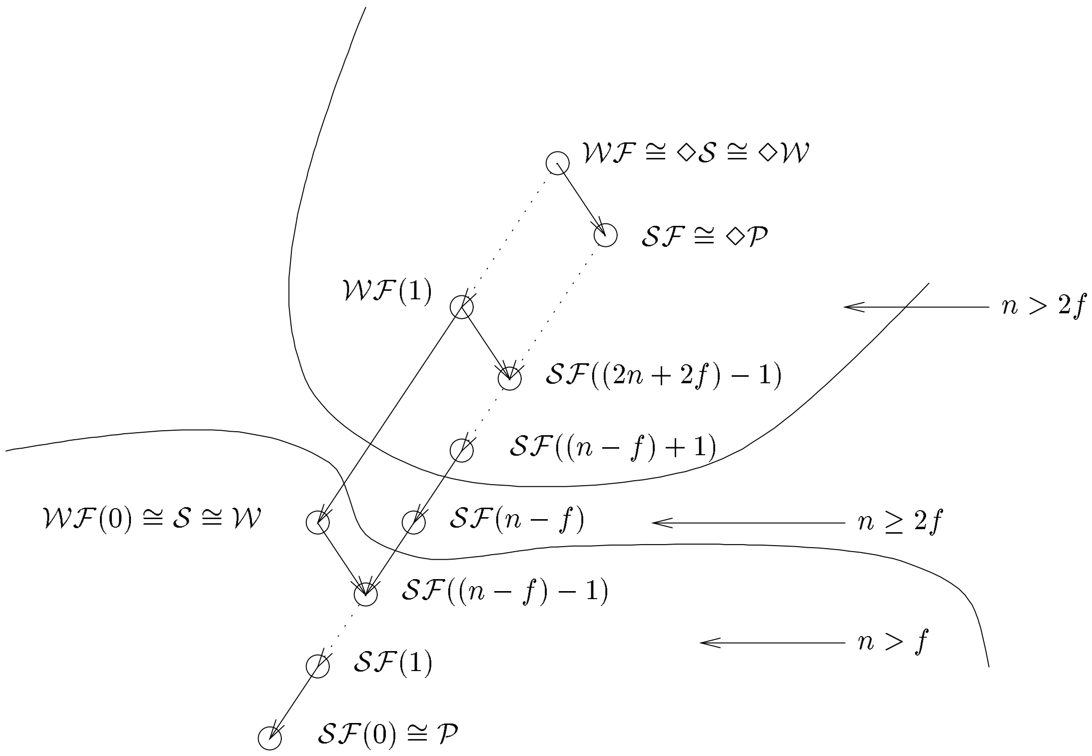

Figure 2: The lattice of failure detectors (ordered by reducibility). This also shows the requirement on f for solving Spontaneous Consensus with these failure detectors.

This hierarchy is illustrated in Figure 2 where the different classes of failure detectors are ordered with respect to failure detector reducibility. In particular $\mathcal{SF}(0) \sqsubseteq \mathcal{SF}(1) \sqsubseteq \ldots \mathcal{SF}(k) \sqsubseteq \mathcal{SF}(k+1) \sqsubseteq \ldots \sqsubseteq \mathcal{SF}$. A similar order holds for the $\mathcal{WF}(k)$s.

The six classes of failure detectors that we considered in Sections 3.2 and 3.3 are equivalent to four members of the hierarchy. In particular we can show that:

$$\mathcal{P} \cong \mathcal{SF}(0)$$
$$\mathcal{W} \cong \mathcal{S} \cong \mathcal{WF}(0)$$
$$\Diamond\mathcal{P} \cong \mathcal{SF}$$
$$\Diamond\mathcal{W} \cong \Diamond\mathcal{S} \cong \mathcal{WF}$$

For example, the conversion of a failure detector $D \in \mathcal{WF}$ to a failure detector $S \in \Diamond\mathcal{S}$ is achieved by Algorithm 4. Other conversions are similar or straightforward and are left to the reader.

From Theorem 2 and 5 we have:

Corollary 5.3: Given any failure detector in $\mathcal{WF}(0)$, Spontaneous Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < n$.

From Theorem 4 and 5 we have:

Corollary 5.4: Given any failure detector in $\mathcal{WF}$, Spontaneous Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < \frac{n}{2}$.

A Lower Bound on Fault-Tolerance

In Sections 6 and 8 we showed that given any failure detector with *weak accuracy*, we can solve both Consensus and Atomic Broadcast for any number of process failures ($f < n$). However, given a failure detector with only *eventual weak accuracy*, our solutions to Consensus and Atomic Broadcast require $f < \frac{n}{2}$. In this section, we show that this requirement is necessary. To prove this lower bound, we first refine the model of asynchronous systems that we consider.

An asynchronous system is defined as a set of *runs*. Each run is sequence of *atomic events* executed by the processes in the system. There are two types of atomic events:

- A *communication event* at a process p consists of three successive components: a *receive* component in which p may receive a single message, say m, a *computation* component in which p may change state based on m, and a *send* component in which p may send a message based on its new state.

- A *query event* consists of querying the failure detector, getting back a set of suspected processes, and changing state accordingly (thus, a query event executed by processes p at time t in a run σ returns $D_p(t, \sigma)$).

Theorem 6: There is a failure detector D in $\mathcal{SF}((n-f)+1)$ such that, given D, there is no algorithm A which solves Consensus in asynchronous systems with $\frac{n}{2} \leq f$.
Proof: [*sketch*] We construct a failure detector $D \in \mathcal{SF}((n-f)+1)$ such that with every algorithm A, there is a run σ_A of A with D such that σ_A does not satisfy the specifications of Consensus. Partition the set of processes Q into two sets Q_0 and Q_1 of size f and $n-f$ respectively. Let $q_0 \in Q_0$ and $q_1 \in Q_1$. Consider any algorithm A, and the following two runs of A with the corresponding behaviour of D:

- Run σ_0: All processes in Q_0 have initial value 0, and all processes in Q_1 have initial value 1. All processes in Q_0 are correct, while those in Q_1 crash at the beginning of the run (this is possible since $n-f \leq f$). The behaviour of D is as follows. Process q_0 always suspects all processes in Q_1, i.e., for all times t, $D_{q_0}(t, \sigma_0) = Q_1$. No other process suspects any process, i.e., for all t and all $q \neq q_0$, $D_q(t, \sigma_1) = \emptyset$. In this run, it is clear that D satisfies the specification of $\mathcal{SF}((n-f)+1)$.

- Run σ_1: As in σ_0, all processes in Q_0 have initial value 0, and all processes in Q_1 have initial value 1. All processes in Q_1 are correct, while those in Q_0 crash at the beginning of the run. Process q_1 always suspects all processes in Q_0, and no other process suspects any process. D also satisfies the specification of $\mathcal{SF}((n-f)+1)$ in this run.

Assume, without loss of generality, that both σ_0 and σ_1 satisfy the specifications of Consensus. Let t_0 be the time at which q_0 decides in σ_0, and let t_1 be the time at which q_1 decides in σ_1. There are three possible cases — in each case we show that we can define a run σ_A of algorithm A, and the corresponding behaviour of the failure detector $D \in \mathcal{SF}((n-f)+1)$, such that σ_A violates the specifications of Consensus.

1. In σ_0, q_0 decides 1. Let σ_A be a run identical to σ_0 except that all processes in Q_1 start with 0. Since in both σ_0 and σ_1 the processes in Q_1 crash right from the beginning, these two runs are indistinguishable to q_0. Thus q_0 decides 1 in σ_A (as it did in σ_0), thereby violating the *validity* condition of Consensus.

2. In σ_1, q_1 decides 0. This case is similar to Case 1.

3. In σ_0, q_0 decides 0, and in σ_1, q_1 decides 1. Construct σ_A and D as follows. The initial values of all processes in Q_0 is 0, and the initial value of all processes in Q_1 is 1. All

messages from processes in Q_0 to those in Q_1 and vice-versa, are delayed until time $t_0 + t_1$. Until time t_0, σ_A is identical to σ_0, except that the processes in Q_1 do not crash, they are only "very slow" and do not execute any event before time t_0. Thus, until time t_0, q_0 cannot distinguish between σ_0 and σ_A, and it decides 0 at time t_0 in σ_A (as it did in σ_0). Note that by time t_0, the failure detector D made $n - f$ mistakes in σ_A: q_0 erroneously suspected that all processes in Q_1 crashed (while they were only slow).

From time t_0, the construction of σ_A and D continues as follows.

(a) At time t_0, all processes in Q_0, except q_0, crash.

(b) From time t_0 to time $t_0 + t_1$, q_1 suspects all processes in Q_0, i.e., $\forall t \in [t_0, t_0 + t_1]$: $D_{q_1}(\sigma_A, t) = Q_0$. By suspecting all the processes in Q_0, including q_0, the failure detector at process q_1 makes one mistake. Thus by time $t_0 + t_1$, D makes a total of $n + 1 - f$ mistakes.

(c) At time t_0, processes in Q_1 "wake up." From time t_0 to time $t_0 + t_1$ they execute exactly as they did in σ_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 σ_A, q_1 decides 1 (as it did at time t_1 in σ_1). So q_0 and q_1 decide differently in σ_A, and this violates the *agreement* condition of Consensus.

(d) From time $t_0 + t_1$, all processes in $Q_1 \cup \{q_0\}$ suspect exactly all the processes that have actually crashed. This ensures that D satisfies the specifications of $\mathcal{SF}((n-f)+1)$ in σ_A. □

Theorem 7: There is a failure detector $D \in \mathcal{SF}((n-f)+1)$ such that, given D, there is no algorithm A which solves Atomic Broadcast in asynchronous systems with $\frac{n}{2} \leq f$.
Proof: From Theorem 6 and the equivalence of Consensus and Atomic Broadcast. □

Corollary 7.1: There is a failure detector D in $\diamond\mathcal{P}$ such that, given D, there is no algorithm A which solves Consensus (Atomic Broadcast) in asynchronous systems with $\frac{n}{2} \leq f$.

This corollary is also a direct consequence of Theorem 4.3 in [DLS88].

Appendix B: A time-out based implementation of an Eventually Strong Failure Detector

In this section, we describe a possible implementation of an Eventually Strong Failure Detector (see Algorithm 7). This implementation assumes a time-out mechanism with the following properties:

1. If any process p crashes then *eventually* some correct process times-out on p.

2. There is a *correct* process p and a time after which no premature time-out on p occurs.

These are reasonable assumptions for most practical systems.

Theorem 8: In a system with the above time-out mechanism, Algorithm 7 implements an Eventually Strong Failure Detector.

Proof: Algorithm 7 is similar to Algorithm 4. Instead of using an Eventually Weak Failure Detector to try to detect crashes, as Algorithm 4 does, Algorithm 7 uses the given time-out mechanism. The proof of correctness is similar to Theorem 3. □

It is easy to implement a time-out mechanism with the above two properties in the two models of partial synchrony defined in [DLS88] (the basic technique is to increase the length of a time-out every time a premature time-out is detected). So by Theorem 4, 5, and 8, we can solve Consensus and Atomic Broadcast in these two models.

{*The algorithm for a process p*}

$S_p \leftarrow \emptyset$ {*Failure detector module constructed by p*}

for all $q, r \in Q$

$\quad suspected_p(r, q) \leftarrow 0$ {*Number of times q timed out on r according to p*}

$\quad refuted_p(r, q) \leftarrow 0$ {*Number of times r refuted q according to p*}

cobegin

|| *Task 1:* **repeat periodically**

for all processes $r \notin S_p$

initiate time-out mechanism to detect if r has crashed

|| *Task 2:* **when** time-out on some process r

broadcast$(p, suspects, r, refuted_p(r, p) + 1)$

|| *Task 3:* **when** *deliver*$(q, suspects, r, j)$

$suspected_p(r, q) \leftarrow \max(suspected_p(r, q), j)$

if $p = r$ **then** *broadcast*$(p, refutes, q, j)$

|| *Task 4:* **when** *deliver*$(r, refutes, q, j)$

$refuted_p(r, q) \leftarrow \max(refuted_p(r, q), j)$

|| *Task 5:* **repeat forever**

for all processes r

if $\exists q : suspected_p(r, q) > refuted_p(r, q)$ **then** $S_p \leftarrow S_p \cup \{r\}$

else $S_p \leftarrow S_p - \{r\}$

coend

Algorithm 7: A time-out based implementation of an Eventually Strong Failure Detector.