Fail-Aware Failure Detectors

Christof Fetzer and Flaviu Cristian
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093−0114*
http://www-cse.ucsd.edu/users/{cfetzer, flaviu}
CSE96-475 (Short Version)

Abstract

In existing asynchronous distributed systems it is impossible to implement failure detectors which are perfect, i.e. they only suspect crashed processes and eventually suspect all crashed processes. Some recent research has however proposed that any "reasonable" failure detector for solving the election problem must be perfect. We address this problem by introducing two new classes of fail-aware *failure detectors that are 1) implementable in existing asynchronous distributed systems, 2) not necessarily perfect, and 3) can be used to solve the election problem. In particular, we show that there exists a fail-aware failure detector that allows to solve the election problem and which is strictly weaker than a Perfect failure detector.*

1 Introduction

Failure detectors [3] are a mechanism for adding synchronism to the *time-free* asynchronous system model [9]. Processes of such systems have access to local failure detector modules which maintain a set of processes that are suspected to have crashed. Failure detectors typically satisfy certain properties that allow certain well-known problems, such as consensus, election and atomic broadcast to be solved. These problems are otherwise not solvable in time-free systems [9]. Therefore, failure detectors are in general not implementable in the time-free model. However, existing asynchronous systems, in particular, networks of workstations, have enough "synchronism" to allow the implementation of failure detectors which allow a deterministic solution of problems such as consensus, election and atomic broadcast.

Failure detectors are typically required to satisfy at least a completeness and an accuracy property: (1) completeness properties state how the crash of a process p has to result in the suspicion that p has crashed, and (2) accuracy properties state when failure detectors are allowed to suspect a process to have crashed. For example, "strong completeness" requires that each correct process will eventually suspect all crashed processes, and "strong accuracy" requires that no process is suspected before it has crashed [1]. A *Perfect* failure detector satisfies strong completeness and strong accuracy.

The (highly available leader) election problem [4] requires that at any point in real-time there be at most one leader and for any time s there exists a time $t > s$ for which there is a leader. A recent article [10] proves that a Perfect failure detector is the weakest "reasonable" failure detector which allows a deterministic solution of the election problem, where "reasonable" means a failure detector that satisfies certain symmetry conditions. Our experience indicates that (1) a Perfect failure detector is not implementable in existing asynchronous systems (see Section 7), and (2) the election problem is solvable in these systems [6]. To clarify this apparent contradiction, we (a) introduce two *fail-awareness* properties for failure detectors, (b) use them to define two new classes of *fail-aware* failure detectors, (c) show how a fail-aware failure detector can be used to provide a deterministic solution for the election problem while still being implementable in existing *timed* asynchronous systems, i.e. asynchronous systems in which processes have access to local hardware clocks, and (d) show that there exists a fail-aware failure detector which is strictly weaker than a Perfect failure detector.

The main idea of fail-aware failure detectors is that a process p's failure detector module knows when "too many" processes might erroneously suspect that p has crashed and notifies p of that. This property can be used to ensure that at any time there is at most one leader: when p is the current leader, and p is suspected by a majority of processes (that could elect a new leader), p's failure detector module suspects p and this "self-suspicion" lets p know that it has been demoted to make place for a new leader.

*To appear in the Proceedings of the 15th Symposium on Reliable Distributed Systems, October 23-25, 1996, Niagara-on-the-Lake, Canada. This research was partially supported by a grant from the Air Force Office of Scientific Research.

2 Related Work

Fail-awareness [8] is a general concept to extend safety properties of a fault-tolerant synchronous service by an *exception indicator* so that the new service becomes implementable in timed asynchronous systems [5]. The idea is that the indicator tells a server and its clients whether a safety property currently holds or if it might be violated because the system has suffered "too many performance failures". In case of failure detectors we can think of the accuracy properties as safety properties which could be violated when "too many performance failures" have occurred or the system is partitioned. For example, strong accuracy ensures that a non-crashed process is not suspected by any process. The strong fail-awareness property tells a non-crashed process p if other processes might violate the accuracy property, i.e. it lets p know whether another process might suspect p. We formulate two fail-awareness properties independently of the accuracy properties. This allows us to combine previously introduced accuracy and completeness properties [1] with a fail-aware property to define new fail-aware failure detector classes.

Sabel and Marzullo prove that a Perfect failure detector is the weakest failure detector to solve the election problem in the class of "reasonable" failure detectors [10]. This class contains all failure detectors which satisfy some given *symmetry conditions*. In the newest version of their paper they explicitly exclude the case that a process p can use its local failure detector module to derive that other processes might wrongly suspect p. Fail-aware failure detectors provide such a knowledge and therefore allow a solution of the election problem even though some of them are strictly weaker than a Perfect failure detector.

Failure detectors [3, 2, 1] are not the only basic distributed service that can be used to solve the election problem in asynchronous systems. General purpose asynchronous group membership protocols such as the one-round and the three-round protocols of [5] can be used to provide deterministic solutions to the election problem when the underlying process and communication system satisfies certain "progress assumptions" [6] (see also section 6.1).

3 Model

Failure detectors are a mechanism to introduce additional synchronism into the time-free model. The implementation of a failure detector has therefore to be based on a model with stronger properties than the time-free model. Thus, in this paper we use two models: the *time-free* model and the *timed* asynchronous system model extended by a "progress assumption" (see Figure 1). We will define fail-aware failure detectors for the time-free model [9]. In Section 5 we will show how to solve the election problem in the time-free model using a fail-aware failure detector. In Section 6 we will show how to implement a fail-aware failure detector in the timed model [5] extended by a progress assumption [6]. The main differences between the two models is that the timed model assumes local hardware clocks and defines process and message performance failures (see Section 6.1). In Section 7 we use both models to show that there exists a fail-aware failure detector strictly weaker than a Perfect failure detector that allows the solution of the election problem.

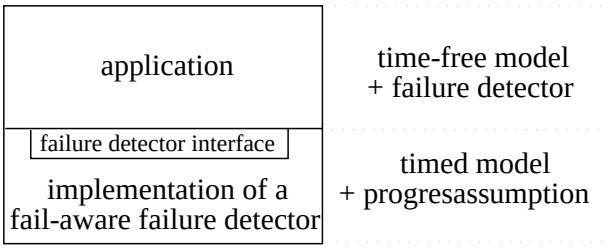

Figure 1. We use the timed model to show how to implement a fail-aware failure detector. Applications using fail-aware failure detectors are designed for the time-free model.

For self-containment we give a brief overview of the time-free model and some known failure detector properties. We use the same notation and essentially the same terminology as [1]. The time-free model does not bound the message transmission delay, the time to execute one step, nor does it mention time, clocks or time-outs. Processes are connected by reliable communication channels. The only failures that can occur are crashes of processes. A process is called *correct* when it never crashes. Processes do not recover from crashes. The set of processes is denoted by Π. Constant n denotes the number of processes, i.e. $n \stackrel{\Delta}{=} |\Pi|$.

A failure detector is a distributed service implemented by a set of local *failure detector modules*, one for each process. Each module maintains a set of processes which it currently suspects to have crashed. Processes can be added or removed from this set. The set of suspected processes is constrained by properties such as *completeness* and *accuracy* [1]. *Strong completeness* requires that every process that crashes be eventually suspected in a permanent manner by all non-crashed processes. *Weak completeness* requires that every process that crashes be eventually suspected in a permanent manner by some correct process. *Strong accuracy* requires that no process be suspected before it crashes and *weak accuracy* requires that some correct process be never suspected. *Eventual strong accuracy* requires the existence of a time after which correct processes are not suspected. *Eventual weak accuracy* requires the existence of a time after which some correct process is never suspected.

3.1 Failure Detector Classes

A failure detector class is a set of all failure detectors that satisfy some given properties. The class of *Perfect failure detectors* P contains all failure detectors which satisfy the strong completeness and the strong accuracy properties. The class of *Eventually Strong failure detectors* $\diamond S$ satisfies

the strong completeness and the eventual weak accuracy properties.

A *reduction algorithm* $T_{D \to D'}$ transforms any implementation of a failure detector D into an implementation of a failure detector D'. A failure detector D' is *weaker* than a failure detector D iff there exists a reduction algorithm $T_{D \to D'}$ that transforms D into D'. Let $\diamond S_0$ denote the "weakest" failure detector in $\diamond S$, i.e. $\diamond S_0$ satisfies the properties of strong completeness and eventual weak accuracy but no other properties. [2] shows that $\diamond S_0$ is a weakest failure detector that allows a deterministic solution of the consensus problem. By P_0 we denote the "weakest" failure detector in P.

4 Fail-Awareness Properties

We introduce two new properties for failure detectors: *weak* and *strong fail-awareness*. In what follows, we use the expression "process p suspects process q" to say that "p's failure detector module suspects q". In particular, we use the expression "p suspects itself" to say that "p's failure detector module suspects p".

A fail-awareness property requires that a process p's failure detector module knows whenever p is suspected by, say, k or more processes. The module notifies p of the occurrence of such an event by suspecting p itself. A fail-awareness property implies that whenever p is not suspected by its local failure detector module, less than k processes do suspect p. However, when a process p suspects itself, p might or might not be suspected by k or more failure detector modules.

Strong fail-awareness requires that a process p suspects itself as soon as some process q suspects p, i.e. $k = 1$. For example, consider a system with two processes p and q (see Figure 2). When q starts suspecting p at some time t and stops suspecting p at time u, then process p has to suspect itself during interval $[t, u]$.

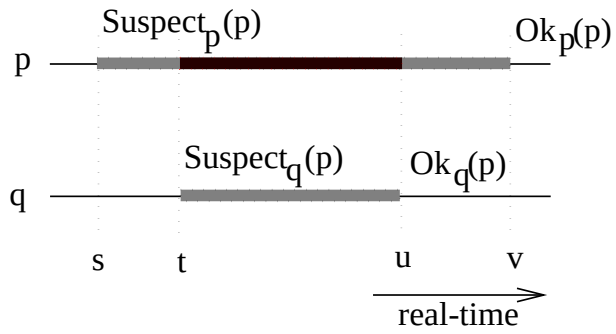

Figure 2. Strong fail-awareness requires that a process p suspects itself as soon as any other process q suspects p.

Weak fail-awareness requires that a failure detector module suspects its process p whenever a majority of processes suspect p, i.e. $k = \lceil \frac{n+1}{2} \rceil$. Consider a system with three processes p, q and r (see Figure 3). Let s, t, u, and v be four points in real-time such that $s \leq t \leq u \leq v$. When r suspects p in interval $[s, u]$ and q suspects p in interval $[t, v]$, then p has to suspect itself at least in interval $[t, u]$ when both q and r suspect p. However, p is not required to suspect itself in intervals $[s, t]$ and $[u, v]$.

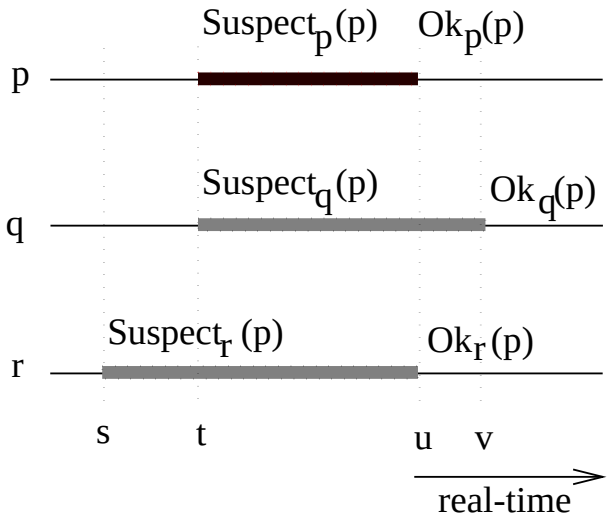

Figure 3. Weak fail-awareness requires that a process p suspects itself when a majority of processes suspect p.

Note that the two fail-awareness properties do not require that some process eventually does not suspect itself. However, the combination of a fail-awareness property with an accuracy property enforces that not all processes do suspect themselves all the time.

Formally, we can define the two fail-awareness properties as follows. A failure detector D satisfies the *weak fail-awareness* property iff for any time t and any process p that is not crashed at t, if p is suspected by a majority of failure detectors at time t, then p's failure detector suspects p at t. In the terminology of [1] this property can be stated as follows. For any failure pattern F, for any time t, any valid failure detector history H for F, and any process p not crashed at t, if the non-crashed processes that suspect p at t form a majority, then p also suspects itself at t:

```
∀ F, ∀ t, ∀ H ∈ D(F), ∀ p ∈ Π-F(t):
 |{ q : q ∈ Π-F(t)
        ∧ p ∈ H(q,t)}| > n/2 ⇒ p ∈ H(p,t)
```

A failure detector D satisfies the *strong fail-awareness* property iff for any time t and for any two processes p and q that are not crashed at t, when q suspects p at t, then p suspects itself at t:

```
∀ F, ∀ t, ∀ H ∈ D(F), ∀ p,q ∈ Π-F(t):
  p ∈ H(q,t) ⇒ p ∈ H(p,t)
```

4.1 Fail-Aware Failure Detectors

We call a failure detector *fail-aware* if it satisfies the weak or the strong fail-awareness property. In this paper we are particularly interested in two new failure detector classes: $\diamond S^{WF}$ and $\diamond S^{SF}$. Class $\diamond S^{WF}$ contains all failure detectors that satisfy strong completeness, eventual weak accuracy, and weak fail-awareness. Class $\diamond S^{SF}$ contains all failure detectors that satisfy strong completeness, eventual weak accuracy, and strong fail-awareness. Each failure detector in $\diamond S^{WF}$ or $\diamond S^{SF}$ guarantees that there exists a process p which is eventually never suspected by any process, and thus, p eventually never suspects itself. We show in Section 5 that any failure detector in $\diamond S^{WF}$ can solve the election problem when a majority of processes are correct. All failure detectors in $\diamond S^{SF}$ allow a deterministic solution of the election problem when at least one process is correct.

Chandra et al. [1] proposed a transformation algorithm that transforms a failure detector FD_W that satisfies the weak completeness property into a failure detector FD_S that satisfies the strong completeness property. However, this transformation does not preserve the weak or the strong fail-awareness property. We therefore only consider failure detector classes $\diamond S^{SF}$ and $\diamond S^{WF}$ that satisfy the strong completeness instead of the weak completeness property.

5 Election

The election problem requires that

(S) at any time t there exists at most one leader, and

(L) for any time s there exists a time $t > s$ for which there exists a leader.

We show how the failure detectors in $\diamond S^{WF}$ and $\diamond S^{SF}$ can be used to solve the election problem. The two election protocols that we provide are *preemptive* in the sense that they can demote a "non-performant" leader to replace it by a new leader. A leader is demoted whenever it is suspected by its failure detector. However, the two protocols guarantee that eventually one process stays leader permanently because the eventual weak accuracy property guarantees that eventually one process is never suspected. Note that election protocols based on group membership services [5] are also preemptive because they can demote a "slow" or "disconnected" leader.

Like the authors of [10], we do require that a process can detect any change in the output of its failure detector module. We therefore assume that the failure detector module of process q maintains for each process p a flag $s_q(p)$ and a version number $v_q(p)$ (see Figure 4):

- flag $s_q(p)$ = true represents the fact that process p is suspected by process q, and

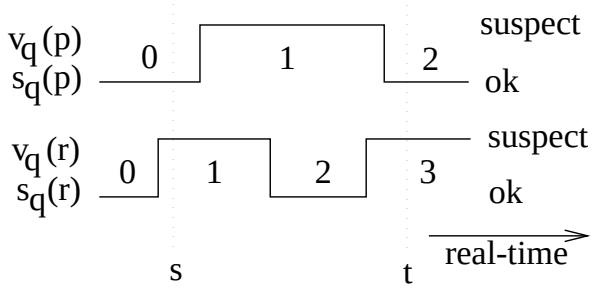

Figure 4. The fail-aware failure detector module of a process q provides for each process p a suspicion flag $s_q(p)$ and a version number $v_q(p)$ that is incremented every time $s_q(p)$ changes its value.

- version number $v_q(p)$ is incremented whenever the value of $s_q(p)$ changes, that is, whenever p is included or excluded from q's set of suspected processes.

Version numbers enable a client q of a failure detector module to determine if a process p was continuously suspected between two of q's samplings of the failure detector module. For example, when process q samples its module for some process p at times s and t (see Figure 4) and if $v_q(p)$ at t is greater than $v_q(p)$ at s, then q has changed it suspicion of p at least once between s and t. Note that [10] assumes instead of version numbers an event oriented mechanism that allows a process to detect any change in the output of its failure detector module.

5.1 Election with $\diamond S^{WF}$

Any failure detector in $\diamond S^{WF}$ can be used to solve the election problem provided a majority of processes are correct. The proposed election protocol has a similar structure like the consensus protocol for $\diamond S$ described in [1]. Both protocols are round based and for each round there is an a priori defined "coordinator". Because the rounds seen by different processes can overlap, there can exist two coordinators at the same time. However, a coordinator can use the weak fail-awareness property to detect when there might be more than one coordinator at a time. The eventual weak accuracy and strong completeness property can be used to guarantee that the rounds are eventually in "synch" in the sense that all non-crashed processes are in the same round and therefore there is only one coordinator. When a coordinator has successfully verified that it is the only coordinator it becomes the leader and stays leader until it crashes or suspects itself.

Our protocol achieves the safety property (S) in the following manner (see Figure 5). For a process p to become leader, it is necessary that p's election is supported by a majority of processes. A process q which has supported p's election will not participate in a new election unless (1) p sends q a "GiveUp"-message indicating that it was demoted, or (2) q learns that a majority of processes suspect p. To guarantee

(S), it is therefore sufficient that p is demoted as soon as a majority of processes might suspect p, that is, p begins to suspect itself. When p does not suspect itself (that is, at most a minority of processes suspect p) and p has not sent its GiveUp-message, no new leader r can be elected. Indeed, if another leader r could be elected before p is demoted (and since any two majorities must have at least one process q in common), there would exist a process q which participated in the elections of both p and r while at the same time knowing that a majority of processes suspect p. This is in contradiction with the fact that p is demoted as soon as a majority of processes suspect p.

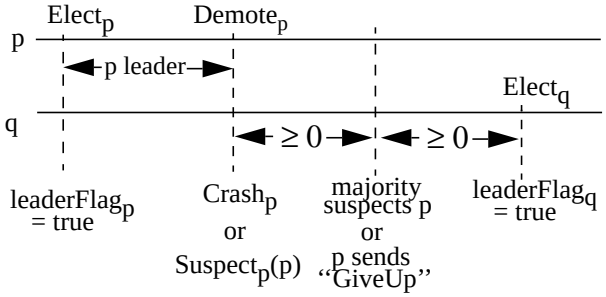

Figure 5. A process p becomes the leader ($Elect_p$) whenever it sets its variable $leaderFlag_p$ to true. It stays leader until it suspects itself or it crashes.

The implementation of the liveness property (L) is based on the following ideas. Processes try to elect a leader in a round based fashion. Only the "coordinator" p of a round R is eligible to become the leader. We therefore call p the *eligible leader* (in R). A process q which has supported p's election in R starts a new round $R+1$ when either (1) q has received a GiveUp-message from p, or (2) q learns that a majority of processes suspect p. By using a sequence of "consistent snapshots", each non-crashed process eventually learns that a crashed leader is suspected by a majority of processes. From the above "new round start rule", it can therefore be derived that when p has crashed or has been demoted, all non-crashed processes eventually start a new round $R+1$. Therefore, some process r different from p becomes the eligible leader in $R+1$. The eventual weak accuracy property ensures that, eventually, there exists at least some correct process l which is not suspected by any process. This property ensures that, when no other process happens to become leader permanently (before l), l will succeed in becoming leader permanently.

5.2 Protocol

The pseudo-code of the proposed protocol WE is given in [7] and cannot be given in this paper due to space restriction. The eligible leader is a priori determined in a round robin fashion. Let p be the eligible leader for round R. When process p starts round R and p does not suspect itself, p broadcasts an "election"-message to let all processes know that p wants to become leader. In case p suspects itself in R, p broadcasts a "GiveUp"-message instead of the election-message to let the other processes know that it does not want to become leader. By *broadcasting* we mean that p sends the same message to all processes. The GiveUp-message allows the processes to start a new round knowing that p is not the leader.

Let us assume that two processes p and q are in round R and p is the eligible leader in R. If q receives an election-message from p and q suspects p, then q replies with a "no-support"-message to let p know that it does not support p's election. If q does not suspect p, q replies with a "support"-message that supports p's election. As soon as p has received replies from at least a majority of processes (i.e. $\lceil \frac{n+1}{2} \rceil$), and all these messages are support-messages, p becomes the leader and stays leader as long as p does not suspect itself and p does not crash. When process p suspects itself or when it has received a no-support-message amongst the first $\lceil \frac{n+1}{2} \rceil$ replies to its election message, p broadcasts a GiveUp-message to allow the other processes to start the next round.

A process q which has supported the election of p at time s has to wait until either (C1) q receives a GiveUp-message from p, or (C2) q learns that there exists a time $t > s$ for which a majority of processes suspect p. Note that whenever condition (C2) is true, p cannot be the leader because either p has crashed, or p has suspected itself at t. The protocol uses a sequence of "consistent snapshots" to implement (C2) . When p is demoted and does not crash, it is guaranteed that every correct process will eventually receive p's GiveUp-message. It is therefore sufficient, to ensure (C2) in case p has crashed.

Process q sends "snapshot"-messages in a round based fashion to query all failure detector modules if they suspect the eligible leader p. When a process r receives such a message, it queries its failure detector module about p and returns the result in a "state"-message. In particular, a state-message contains the current suspicion flag $s_r(p)$ and the version number $v_r(p)$ of r's failure detector module. After receiving $\lceil \frac{n+1}{2} \rceil$ replies to a snapshot-message, q starts a new snapshot-round unless it has determined that condition (C2) holds. The intuition behind the determination of (C2) is that when the same majority M of processes p in two successive rounds (i.e. there exists a time t such that the first round finishes before t and the second round is started after t), and their failure detector modules have not changed their output for p between these two rounds, then all processes in M suspect p at t.

Since a process q starts a new snapshot-round as soon as it has received a reply from $\lceil \frac{n+1}{2} \rceil$ processes and more than $\lceil \frac{n+1}{2} \rceil$ processes can reply to a snapshot message, it is possible that the sets of the first $\lceil \frac{n+1}{2} \rceil$ processes that reply to successive snapshot-messages differ. Moreover, when a reply m for snapshot-round SR arrives after snapshot-round SR+1 has started, it is possible that m was sent after $SR+1$ was already started. In other words, there exists not necessarily a time t for which a majority of processes suspect the eligible leader p. The solution to this problem is to "merge" snapshot-rounds (see Figure 6).

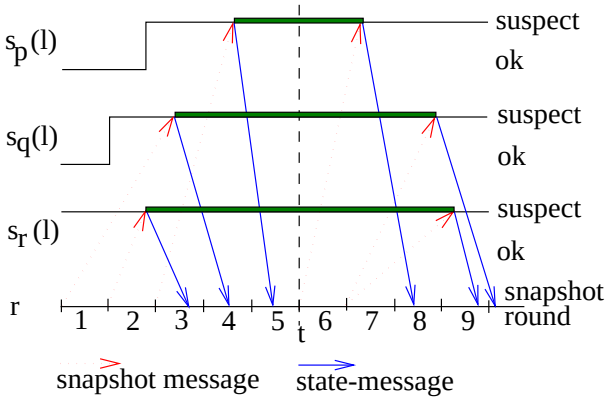

Figure 6. The snapshot algorithm queries all failure detector modules until it finds a majority $M = \{p, q, r\}$ of processes and a time t such that each process in M has sent two state-messages saying that it has constantly suspected the eligible leader l in an interval that includes t. Therefore, l has to suspect itself at t.

Let a state-message also contain the snapshot-round number during which the requesting snapshot-message was sent. Each process r keeps for each process q that suspects the eligible leader l:

- The snapshot-round number, denoted *MinRecStamp[q]*, when r has received the first state-message m with the highest failure detector version number $m.v_q(l)$ from q so far (i.e. all state-messages m' that r has received before m from q contain a smaller failure detector version number for l: $m'.v_q(l) < m.v_q(l)$).
- The highest snapshot round number, denoted *MaxSendStamp[q]*, included in a state message with failure detector version number $m.v_q(l)$ that r has received so far from q, i.e. r sent the snapshot message (that requested this state message) in snapshot round *MaxSendStamp[q]*.

When *MinRecStamp[q]* is smaller than *MaxSendStamp[q]*, then we know that q's failure detector module output for the eligible leader l has not changed between the end of r's snapshot-round $MinRecStamp[q]$ and the start of r's snapshot-round $MaxSendStamp[q]$. Process r performs a search to find a snapshot-round number $threshold$ such that there exists a majority of processes which suspect the eligible leader l at the time snapshot-round $threshold$ was started. When the eligible leader l has crashed, eventually all processes will permanently suspect p. Hence, all failure detector version numbers for process l will eventually stop changing. Array $MinRecStamp$ will therefore eventually stop changing while the entries for all correct processes in array $MaxSendStamp$ will continue to increase. Therefore, eventually process r will succeed to find a majority of processes that suspects l at the start of some snapshot-round $threshold$.

The protocol exports a function *Leader?* that determines iff the calling process is the current leader. The function returns a tuple *(leaderFlag,version)*. The version number returned by *Leader?* is incremented every time the process becomes leader or is demoted. It lets a caller determine iff it was continuously the leader between any two calls of function *Leader?*. Before process p can become leader it queries its failure detector module to make sure that it does not suspect itself. Function *Leader?* also queries the local failure detector module to check if the calling process p has suspected itself since it became leader. In case p has not suspected itself since it became leader, function *Leader?* returns *leaderFlag = true*. Otherwise, p has been demoted and *Leader?* returns *leaderFlag = false*.

Theorem T2: Protocol WE is a correct election protocol.

Proof: see [7].

5.3 Solving Election with $\diamond S^{SF}$

Any failure detector in $\diamond S^{SF}$ can be used to solve election as long as at least one process survives. We propose an election protocol SE which is similar to the protocol for $\diamond S^{WF}$: we modify protocol WE to correctly handle runs in which a majority of processes are crashed. In particular, (1) an eligible leader cannot expect that a majority of processes reply to its election message, and (2) a process q supporting the election of the eligible leader p cannot expect that after p crashes it can find a majority of processes that suspect p. The strong fail-awareness properties however allows to modify the protocol such that processes do not need to rely on replies from a majority of processes.

5.4 Protocol

A process p can stay leader as long as p does not suspect itself. The strong fail-awareness property ensures that p is demoted as soon as one process suspects p. When a process q has supported the election of p, q can therefore start a new round as soon as q suspects p because it knows that at that point p has already been demoted.

In round R the eligible leader p broadcasts an election-message provided it does not suspect itself. Process p has only to wait for support-messages from processes which p does not suspect because (1) any process q suspected by p cannot be leader, and (2.1) before q can become leader, p has to stop suspecting q, and (2.2) q has either to suspect p which implicitly demotes p, or (2.3) q has to receive a support-message from p, however, p does not send support-messages while being leader.

Theorem T3: Protocol SE is a correct election protocol.

Proof: see [7].

6 Weak Fail-Awareness

The goal of this section is to show how to implement a fail-aware failure detector in a timed system. Let $\diamond S_0^{WF}$ denote the weakest failure detector in $\diamond S^{WF}$, that is, let $\diamond S_0^{WF}$ satisfy the properties of strong completeness, eventual weak accuracy, and weak fail-awareness but no other properties. In this section we sketch how $\diamond S_0^{WF}$ can be implemented in timed asynchronous systems [5] provided they satisfy a certain "progress assumption" [6]. The protocol depends upon *fail-aware datagram* and *fail-aware clock synchronization* services [8]. For self-containment, before we describe our protocol for $\diamond S_0^{WF}$, we give a brief overview of the timed asynchronous system model and these two services.

6.1 Timed Systems and Progress Assumptions

The timed asynchronous system model does not guarantee an upper bound on message transmission and process scheduling delays. Nevertheless, it defines two time-out delays: δ for message transmission delays and σ for process scheduling delays. The time-outs are introduced to define *performance failures*. These occur when the transmission delay of a message or a process scheduling delay (i.e. the time a process takes to react to a trigger event) is greater than the associated time-out delay. Processes have access to hardware clocks. Since the drift rate of hardware clocks is bounded by a very small constant (typically of the order of 10^{-4} to 10^{-6}), an upper bound on the error made in measuring real-time intervals is computable. The failure semantics of processes and communication services are crash/performance and omission/performance, respectively. For consistency with [1], in this paper we will assume that crashed processes do not recover.

$\diamond S_0^{WF}$ is not implementable in timed asynchronous systems without making some additional assumptions. The reason for this is that the timed asynchronous system model allows the existence of a run $\mathcal{R}$ in which no process is crashed and all processes form singleton partitions. In other terms, in $\mathcal{R}$ processes cannot communicate with each other. If there would exist a process p that is eventually not suspected by any non-crashed process, we could find a run $\mathcal{R}'$ in which p is crashed, and $\mathcal{R}'$ is for all processes (except p) indistinguishable from $\mathcal{R}$. Therefore, in the timed asynchronous system model it is not possible to satisfy both eventual weak accuracy and strong completeness. We therefore have to introduce an additional assumption to make $\diamond S_0^{WF}$ implementable.

Progress assumptions introduce additional synchronism by asserting that the system will eventually show "synchronous behavior" for a sufficiently long time [6]. Such synchronous behavior is described by a *stability predicate*. In this paper we introduce a new stability predicate (m^+-stable) which is a strengthened version of the *majority-stability* predicate [6]. We say that the system is m^+*-stable* in a time interval $I = [s, u]$ iff there exists a majority of non-crashed processes M such that

1. none of the processes in M suffers a failure during I,
2. a message sent at a time $t \in [s, u - \delta]$ between two processes in M is delivered in time at its destination, i.e. at or before time $t + \delta$, and
3. every message sent during I between two processes p and q is eventually delivered unless p or q are crashed at some point in I.

We have shown in [6] that the first two conditions (which characterize the majority-stability predicate) are sufficient to solve the election problem. However, these two conditions are not sufficient to implement a failure detector that satisfies the eventual weak accuracy and strong completeness properties. This fact can be proven in a way similar to the above sketched proof that $\diamond S_0^{WF}$ is not implementable in timed asynchronous systems because these two conditions still allow some non-crashed process to be partitioned from the remaining processes. To address this problem, we could define a weaker *majority completeness* property which would still allow to solve the election problem and its implementation would not require condition (3): a crashed process is eventually suspected by a majority of processes. An analysis of the election protocol for $\diamond S^{WF}$ (see Section 5) shows that it is correct even for failure detectors that only satisfy the majority instead of the strong completeness property. Below we will propose a failure detector protocol that satisfies the strong completeness property in case the system is eventually always m^+-stable. We will sketch a slight modified version of that protocol that guarantees majority completeness in case the system is eventually always majority-stable[1]. In this paper we emphasize the protocol for strong completeness and hence, will use the m^+-stability predicate instead of the weaker majority-stability predicate.

In the definition of m^+-stable no uniqueness of the majority M is assumed: it is possible that in an interval I there exists two majorities M_1 and M_2 that satisfy the requirements of m^+-stable. For simplicity, we describe the protocol assuming a unique M, i.e. we select one set M and describe the behavior of the protocol for that set M. However, the protocol is correct even when there are multiple majorities that satisfy the requirements of m^+-stable. Processes in M are called m^+*-stable*, while processes in Π-M are called *non-stable*.

To implement $\diamond S_0^{WF}$ we assume the following progress assumption *(PA)*: the system of processes Π eventually becomes permanently m^+-stable, that is, there exists a time s such that the system is m^+-stable in $[s, \infty]$[2]. In what fol-

[1] Note that the even though the majority-stability predicate would allow us to solve the election problem, it does not allow us to implement a reliable unicast service required by the time-free model because even when the system is majority-stable there can exist two non-crashed processes that cannot communicate with each other.

[2] This is not realistic for existing asynchronous distributed systems with

lows we are only interested in one interval $I_0 \stackrel{\Delta}{=} [s+IT, \infty]$, where constant IT is sufficiently long to allow the fail-aware clock synchronization and datagram services to be initialized. Henceforth, we use "stable" to mean "m^+-stable" and implicitly assume I_0 whenever we talk about stability in our fail-aware failure detector protocol.

6.2 Fail-Aware Services

The essential property of a fail-aware datagram service [8] is that messages delivered to a process are tagged as either "standard" (timely) or "exceptional" (slow). Exceptional messages are slow messages that may contain out-of-date information. A fail-aware datagram service defines a constant $\Delta > \delta$ such that when the transmission delay of a message m is at most δ, it delivers m as a *standard message*. When the transmission delay of m is greater than Δ, m is delivered as an *exceptional message*. A message with a transmission delay within $]\delta, \Delta]$ may be delivered as either exceptional or standard. In this section we implicitly assume that all messages are sent by the fail-aware datagram service.

A fail-aware clock synchronization service [8] provides each process p with an indicator S_p which, when true, tells p that its clock C_p is synchronized. A fail-aware clock synchronization service is required to satisfy the following properties: 1) when the indicators of processes p and q are true, the deviation between the clocks of p and q is at most ψ:

$$S_p(t) \wedge S_q(t) \rightarrow |C_p(t) - C_q(t)| \leq \psi$$

2) the indicator of each stable process is true, 3) the drift rate of synchronized clocks is bounded by a very small constant, and 4) clocks are monotonically increasing. From the above properties, it follows that the deviation between the clock of a non-stable process p and the clock of another process might be greater than ψ. However, in this case p's indicator is false.

6.3 Overview of Protocol

The goal of our protocol is to ensure that (1) eventually no process will suspect any stable process, and (2) each crashed process is eventually suspected by all non-crashed processes. In particular, our first requirement implies that eventually even any non-stable process stops suspecting stable processes. A non-crashed, non-stable process might or might not be suspected by other processes. Requirement (1) and progress assumption (PA) imply that there exists at least one stable process which is eventually never suspected by any other process.

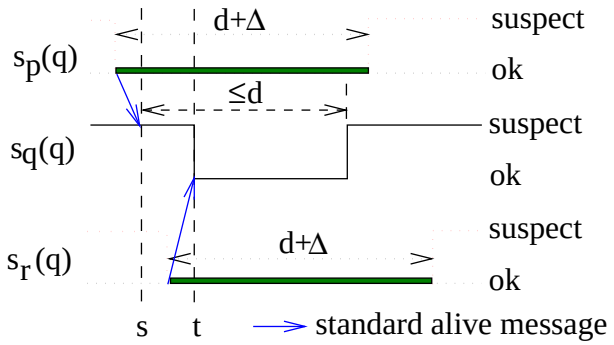

Figure 7. When process q receives a standard alive message at s from a process p such that q is in the message's alive set, then q knows that p will not suspect q for at least d time units after s. Hence, q can stop suspecting itself as long as a majority of processes have guaranteed that they do not suspect q.

The weak fail-awareness property is implemented using periodic broadcasts of alive-messages. By *broadcast* we mean in this section that p sends the same message to all processes using the fail-aware datagram service. Each process q maintains a set $Alive_q$ which contains all processes from which q has "recently" received a standard alive-message. The set $Alive_q$ is piggy-backed on any alive-message that q sends. Because a stable process p receives periodically standard alive-messages from all stable processes, $Alive_p$ contains at least all stable processes. The protocol ensures that any stable process q receives a standard alive-message from any other stable process p at least every d time units. When q receives such a message (see Figure 7), it knows that 1) p is alive, and 2) p will (by convention) not suspect any process in $Alive_p$ for at least $d + \Delta$ time units. The broadcast of alive messages is coordinated using the fail-aware clock synchronization service. Each stable process q will therefore always know that all stable processes will not suspect q, and hence, q will not suspect itself.

An important requirement in the proposed election protocols is that a process can detect any change in the output of its failure detector module. The fail-aware failure detector module of a process q therefore provides for each process p a flag $s_q(p)$ and a version number $v_q(p)$. (see Section 5 for more details).

6.4 Protocol Details

The pseudo-code of the proposed protocol WF is given in [7] and cannot be given in this paper due to space restrictions. The protocol is based on the broadcast of

respect to the requirement that the system will be permanently m^+-stable. We nevertheless introduce it, as was done in [1], to avoid talking about time constants *above* the failure detector abstraction layer, yet to be able to express that the interval is "sufficiently long". Realistically, one could assume that for any time s there exists a time $t > s$ such that the system is stable in interval $[t, t + L]$, where L is a time constant that is "sufficiently long" to elect a new leader. The election protocols proposed in this paper would still be correct under such a weaker progress assumption, that makes sense in a timed asynchronous system model, but does not in the time-free model, where one cannot talk about time constants.

alive-messages in a round based fashion. The fail-aware datagram service is used to discard out-dated information in alive-messages. The start of a round is determined by the fail-aware clock synchronization service. Only when the clock of a process p is synchronized, it sends alive-messages.

The protocol has to ensure that no process q can suspect a stable process p even when p and q cannot communicate in a timely fashion with each other (otherwise the eventual weak accuracy property could be violated). To ensure this, process q is allowed to suspect a process p only when q knows that a majority of processes do not include p in their alive-messages. Because all stable processes include all stable processes in their alive-messages, eventually no process can suspect a stable process.

An important implementation detail is how protocol WF can guarantee the fail-awareness property even when process performance failures occur during the execution of WF. In particular, consider that p's failure detector module knows that a majority of processes will not suspect p in the current round (represented by variable $round$) which lasts at least as long as p's hardware clock shows a value of at most, say, $nextUpdate$. When protocol WF cannot collect alive-message from a majority saying that they will not suspect p during $round + 1$, p has to suspect itself during $round + 1$. When process p asks if its failure detector module suspects p, then the module reads p's hardware clock to see if it already shows a value greater than $nextUpdate$. In case this is true, a performance failure must have occurred and the module will therefore suspect p.

A slight modified version of this protocol can ensure the eventually weak accuracy, the weak fail-awareness, and the majority completeness property in case the system is eventually always majority-stable. In this version, a process which suspects itself does not suspect any other process. Since eventually all majority-stable processes do not suspect themselves, these processes will eventually suspect all crashed processes permanently. In other words, the majority completeness property is satisfied. However, this version does not necessarily satisfy the strong completeness property.

Theorem T1: Protocol WF satisfies the weak fail-awareness, eventual weak accuracy, and the strong completeness properties.
Proof: The proof is given in [7].

7 P_0 versus $\diamond S_0^{WF}$

In this section we show that $\diamond S_0^{WF}$ is strictly weaker than any Perfect failure detector: $\diamond S_0^{WF}$ can be reduced to P_0, but P_0 cannot be reduced to $\diamond S_0^{WF}$. Failure detector P_0 satisfies all properties of $\diamond S_0^{WF}$: P_0 only suspects processes that have crashed and hence, a process has never to suspect itself. We can therefore restrict ourselves to show that there exists no reduction algorithm that transforms $\diamond S_0^{WF}$ into a Perfect failure detector.

The fact that $\diamond S_0^{WF}$ is strictly weaker than P_0 can be explained as follows. P_0 has to be able to decide correctly if a process is just "slow" or crashed. $\diamond S_0^{WF}$ does not have to make this decision: it just suspects a "slow" or crashed process p because a "slow" process p can detect when it is "slow" and then suspect itself. In particular, $\diamond S_0^{WF}$ can be implemented in some systems that do not allow to distinguish between a crashed and a "slow" process while P is not implementable in these systems. When there would exist a reduction algorithm that transforms $\diamond S_0^{WF}$ into P_0, this algorithm would allow to implement P in such systems. In other terms, when we show that there exists a system that allows the implementation of $\diamond S_0^{WF}$ but not P_0, we can prove by contradiction that $\diamond S_0^{WF}$ is strictly weaker than P_0.

In the timed asynchronous system model we can give "slow" a precise meaning: a process is "slow" when it is not stable. We first show that in eventually stable timed asynchronous systems a process cannot correctly decide if a remote process is just slow or crashed. In other words, we show that P_0 is not implementable in these systems. We already know that $\diamond S_0^{WF}$ is implementable in eventually stable timed asynchronous systems. Second, we show by contradiction that $\diamond S_0^{WF}$ is strictly weaker than P_0: when there would exist a reduction algorithm that transforms $\diamond S_0^{WF}$ to P_0, this algorithm could be used to implement P_0 in eventually stable timed asynchronous systems.

The reason why P_0 is not implementable in eventually stable systems is the following. We can select a run $\mathcal{R}$ in which there exists a process p which is not stable. In particular, in run $\mathcal{R}$ we can delay all messages between p and any other process for an arbitrarily long time. Eventually all other, non-crashed processes have to suspect p to ensure the strong completeness property. Otherwise, we could find a from $\mathcal{R}$ indistinguishable run $\mathcal{R}'$ in which p is crashed but not suspected by some non-crashed process. In other words, in an eventually stable system one can implement the strong completeness or the strong accuracy property, but not both.

7.1 Impossibility Proof

We first show that in eventually stable timed systems a process cannot always correctly decide if another process is crashed or just slow.

Theorem T4: No Perfect failure detector is implementable in eventually stable timed systems.
Informal Proof: We prove this theorem by contradiction. Let us assume that P_0 would be implementable in all eventually stable systems. We consider a system which contains at least 3 processes, i.e. we can select runs in which at least one non-crashed process p can be non-stable. We iteratively construct a run such that either the strong completeness or the strong accuracy property is violated. The idea is to select a run $\mathcal{R}$ in which all messages sent from p to any other process are delayed until at least one non-crashed process suspects p. Let us first consider the case that in run $\mathcal{R}$ no process would ever suspect p. This implies that in run $\mathcal{R}$ these delayed messages suffer omission failures and $\mathcal{R}$ is

therefore not valid because the progress assumption (PA) requires that eventually all messages sent between two non-crashed processes are delivered. However, we can select a valid run $\mathcal{R}'$ which is indistinguishable from $\mathcal{R}$ for all processes (except p) and in $\mathcal{R}'$ process p is crashed. Therefore, in run $\mathcal{R}'$ the strong completeness condition is not satisfied.

Second, we consider the case that some process q eventually suspects p in $\mathcal{R}$. After q suspects p, we deliver all delayed messages to make $\mathcal{R}$ valid. Thus, in run $\mathcal{R}$ the strong accuracy condition is violated. Both cases are a contradiction to our initial assumption that P_0 would be implementable in all eventually stable systems, hence theorem T4 holds. □

We can use theorem T4 to show that a Perfect failure detector is not reducible to $\diamond S_0^{WF}$ because $\diamond S_0^{WF}$ is implementable in eventually stable systems but not P_0. Let us first show the following lemma.

Lemma L1: Any reduction algorithm implementable in a time-free system is implementable in an eventually stable timed system.
Informal Proof: It is sufficient to show that a reliable unicast service is implementable in an eventually stable system, i.e. a service which guarantees that a message sent by a process p to a process q is eventually delivered to q unless p or q eventually crashes. The progress assumption PA ensures that eventually all non-crashed processes can communicate using the basic datagram service without the occurrence of omission failures. Thus, a simple positive acknowledgement based protocol can implement a reliable unicast service using the basic datagram service (in case progress assumption PA holds). □

Theorem T5: P_0 is not reducible to $\diamond S_0^{WF}$.
Informal Proof: We prove this theorem by contradiction. Let us assume that there would exist a reduction algorithm T that transform $\diamond S_0^{WF}$ to P_0. We have shown in Lemma L1 that T is implementable in eventually stable timed systems. Hence, P_0 is implementable in eventually timed systems because we have shown in Section 6 that $\diamond S_0^{WF}$ is implementable in such systems. This is a contradiction to theorem T4. □

8 Conclusion

This paper has addressed the problem that (1) the weakest "reasonable" failure detector to solve election seems to be a Perfect failure detector [10], (2) our experience indicates that a Perfect failure detector is not implementable in existing asynchronous systems, and (3) the election problem is solvable in such systems [6]. We resolve the above problem by introducing two fail-aware failure detector classes that contain failure detectors which are strictly weaker than the weakest Perfect failure detector and we show how fail-aware failure detectors can be used to provide deterministic solutions to the election problem.

For compatibility with [1], we considered a stronger completeness property (strong completeness instead of majority completeness) for our failure detectors than necessary to solve election. This forced us to use a stronger than necessary progress assumption to implement a fail-aware failure detector that allows to solve the election problem. However, the proposed election protocol WE that uses the fail-aware failure detector $\diamond S_0^{WF}$ still works for majority completeness. We also sketched a version of the failure detector protocol WF that works for the weaker progress assumption that the system is eventually majority-stable for a "sufficiently long" time.

References

[1] T. Chandra, V. Hadzilacos, and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*. to appear; also available at ftp.cs.cornell.edu as /pub/chandra/failure.detectors.algorithms.ps.Z.

[2] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158, Aug 1992.

[3] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.

[4] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991. An earlier version in *FTCS-18*, 1988, Kyoto.

[5] F. Cristian and F. Schmuck. Agreeing on processor-group membership in aynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/asyncmembership.ps.Z.

[6] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, Newport Beach, CA, Dec 1995.

[7] C. Fetzer and F. Cristian. Fail-aware failure detectors. Technical Report CSE96-475, UCSD, 1996. Available via anonymous ftp at cs.ucsd.edu as /pub/team/failAwareFD. ps.Z.

[8] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, May 1996.

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

[10] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Feb 1995.