A Hybrid Approach for Building Eventually Accurate Failure Detectors

Achour MOSTEFAOUI*, David POWELL†, Michel RAYNAL*

* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
† LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France
{achour|raynal}@irisa.fr david.powell@laas.fr

Abstract

Unreliable failure detectors introduced by Chandra and Toueg are abstract mechanisms that provide information about process crashes. On the one hand, failure detectors allow a statement of the minimal requirements on process failures that allow solutions to problems that cannot otherwise be solved in purely asynchronous systems. However, on the other hand, they cannot be implemented in such systems: their implementation requires that the underlying distributed system be enriched with additional assumptions.

Classic failure detector implementations rely on additional synchrony assumptions such as partial synchrony. More recently, a new approach for implementing failure detectors has been proposed: it relies on behavioral properties on the flow of messages exchanged. This paper shows that these approaches are not antagonistic and can be advantageously combined. A hybrid protocol (the first to our knowledge) implementing failure detectors with eventual accuracy properties is presented. Interestingly, this protocol benefits from the best of both worlds in the sense that it converges (i.e., provides the required failure detector) as soon as either the system behaves synchronously or the required message exchange pattern is satisfied. This shows that, to expedite convergence, it can be interesting to consider that the underlying system can satisfy several alternative assumptions.

Keywords*: Asynchronous distributed systems, Distributed algorithm, Convergence, Hybrid algorithm, Fault tolerance, Process crash, Unreliable failure detector.*

1 Introduction

Context of the study The design and implementation of reliable applications on top of asynchronous distributed systems prone to process crashes is a difficult and complex task. The main reason for this lies in the impossibility of correctly detecting crashes in the presence of asynchrony. In such a context, some problems become very difficult or even impossible to solve. The most famous of those problems is the *Consensus* problem for which there is no deterministic solution in asynchronous distributed systems where processes (even only one) may crash [9].

To overcome this impossibility, Chandra and Toueg introduced the concept of an *Unreliable Failure Detector* [5]. A failure detector can be seen as an oracle [15, 20] made up of a set of modules, each associated with a process. The failure detector module attached to a process provides it with a list of processes it suspects of having crashed. A failure detector module can make mistakes (whence the epithet "unreliable") by not suspecting a crashed process or by erroneously suspecting a correct one. In their seminal paper [5], Chandra and Toueg introduced several classes of failure detectors. A class is defined by two abstract properties, namely a *Completeness* property and an *Accuracy* property. Completeness is on the actual detection of crashes, while accuracy restricts erroneous suspicions. As an example, let us consider the class of failure detectors denoted $\diamond\mathcal{S}$. It includes all failure detectors such that (1) eventually, every crashed process is permanently suspected by every correct process[1], and (2) there is a correct process that, after some finite but unknown time, is never suspected by the correct processes (accuracy). Interestingly, several protocols that solve the consensus problem in asynchronous distributed systems augmented with a failure detector of the class $\diamond\mathcal{S}$, and including a majority of correct processes, have been designed (e.g., [5, 12]). It has been shown that $\diamond\mathcal{S}$ is the weakest class of failure detectors that allows the consensus problem to be solved [6] (with the additional assumption that a majority of processes are correct).

As defined and advocated by Chandra and Toueg [5],

[1] A *correct* process is a process that does not crash. See Section 2.1.

the failure detector approach is particularly attractive. Failure detectors are not defined in terms of a particular implementation (involving network topology, message delays, local clocks, etc.) but in terms of abstract properties (related to the detection of failures) that allow problems to be solved despite process crashes[2]. The failure detector approach allows a modular decomposition that not only simplifies protocol design but also provides general solutions. More specifically, during a first step, a protocol is designed and proved correct assuming only the properties provided by a failure detector class. Thus, this protocol is not expressed in terms of low-level parameters, but depends only on a well-defined set of abstract properties. The implementation of a failure detector FD of the assumed class can then be addressed independently: additional assumptions can be investigated and the ones which are sufficient to implement FD can be added to the underlying distributed system in order to define an augmented system on top of which FD can be implemented. Thus, FD can be implemented in one way in some context and in another way in another context, according to the particular features of the underlying system. It follows that this layered approach favors the design, the proof and the portability of protocols.

Related work Several works have considered the implementation of some or all of Chandra-Toueg's classes of failure detectors [2, 4, 5, 8, 10, 13, 14, 21]. Basically, these works consider that, eventually, the underlying system behaves in a synchronous way. More precisely, they consider the *partially synchronous system* model [5] which is a generalization of the models proposed in [7]. A partially synchronous system assumes there are bounds on process speeds and message transfer delays, but these bounds are not known and hold only after some finite but unknown time (called *Global Stabilization Time*). The protocols implementing failure detectors in such systems use timeouts and are consequently timer-based. They obey the following principle: using successive approximations, each process dynamically determines a value Δ that eventually becomes an upper bound on transfer delays.

Another approach, that does not rely on the use of timeouts, has recently been introduced in [16]. This approach, which uses explicitly the values of n (the total number of processes) and f (the maximal number of processes that can crash), consists in stating a property on the message exchange pattern that, when satisfied, allows the construction of a failure detector of some class.

[2] As an example different from the consensus problem, we can consider the *Global Data Computation (GDC)* problem (a variant of the *Interactive Consistency* problem [18] in asynchronous distributed systems where processes can commit only crash failures). It appears that *perfect* failure detectors -the ones that never make erroneous suspicions- are both necessary and sufficient to solve GDC [11].

Assuming that each process can broadcast queries and then, for each query, wait for the corresponding responses, we say that a response to a query is a *winning* response if it arrives among the first $(n - f)$ responses to that query (the other responses to that query are called *losing* responses). We are now in order to state, as an example, the following property: "There are a correct process p_i and a set Q of $(f + 1)$ processes such that eventually the response of p_i to each query issued by any $p_j \in Q$ is always a winning response (until -possibly- the crash of p_j)". It is shown in [16] that failure detectors of the class $\diamond\mathcal{S}$ can be implemented when this property is satisfied. This means that it is possible to design a protocol satisfying the completeness and accuracy properties of $\diamond\mathcal{S}$ on top of asynchronous distributed systems satisfying the previous requirement. Interestingly, such a requirement does not involve bounds on communication times (they can be arbitrary). A probabilistic analysis for the case $f = 1$, presented in [16], shows that such a behavioral property on message exchanges is practically always satisfied.

Content of the paper This paper shows that the previous approaches are not antagonistic and can be combined in a *hybrid* protocol implementing *eventual* failure detectors. This protocol benefits from the best of both worlds in that it converges as soon as either the system behaves synchronously, or the required message exchange pattern occurs (finding such a combination of conditions was until now an open problem). While hybrid protocols have been proposed to solve consensus (e.g., [3, 17] use both a failure detector and a random oracle), to our knowledge, this is the first hybrid protocol to implement eventual failure detectors. More explicitly, if only one of the following assumptions is satisfied:

- Partial synchrony,
- Behavioral property on the message exchange pattern,

then, the proposed hybrid protocol implements a failure detector that is eventually accurate. If both assumptions are satisfied, let t_S be the time after which the underlying system behaves synchronously, and t_M be the time after which the behavioral property on the message exchange pattern is satisfied. In that case, the eventual accuracy is obtained from $\min(t_S, t_M)$. Moreover, as we will see in the protocol description, using a behavioral property in addition to partial synchrony can be done at no additional cost. It follows that such a hybrid approach is practically appealing.

Organization of the paper The paper is made up of five sections. Section 2 defines the computation model, and the classes of eventual failure detectors we are interested in. These are denoted $\diamond\mathcal{S}$ (eventually strong failure detectors)

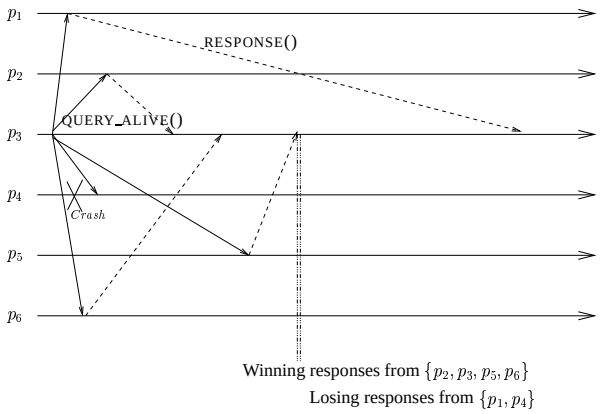

Figure 1. Query/Response Mechanism

and $\diamond\mathcal{P}$ (eventually perfect failure detectors). This section also introduces the additional assumptions we consider for implementing failure detectors of these classes. Then, Section 3 presents a hybrid protocol implementing failure detectors of the class $\diamond\mathcal{S}$. Section 4 considers the case of failure detectors of the class $\diamond\mathcal{P}$. Finally, Section 5 concludes the paper.

2 Computation Model and Assumptions

2.1 Computation Model

Asynchronous distributed system with process crash failures We consider a system consisting of a finite set Π of $n \geq 3$ processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process can fail by *crashing*, i.e., by prematurely halting. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash. A *faulty* process is a process that is not correct. As previously indicated, f denotes the maximum number of processes that can crash ($1 \leq f < n$).

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are assumed to be reliable: they do not create, alter or lose messages. In particular, if p_i sends a message to p_j, then eventually p_j receives that message unless it fails. There is no assumption about the relative speed of processes or message transfer delays (let us observe that channels are not required to be FIFO).

We assume the existence of a global discrete clock. This clock is a fictional device which is not known by the processes; it is only used to state specifications or prove protocol properties. The range $\mathcal{T}$ of clock values is the set of natural numbers.

Query-response mechanism For our purpose (namely, the implementation of failure detectors) we consider that each process is provided with a query-response mechanism. More specifically, any process p_i can broadcast a QUERY_ALIVE() message and then wait for corresponding RESPONSE() messages from $(n-f)$ processes (these are the *winning* responses for that query). The other RESPONSE() messages associated with a query, if any, are systematically discarded (these are the *losing* responses for that query).

A query issued by p_i is *terminated* if p_i has received the $(n-f)$ corresponding responses it was waiting for. We assume that a process issues a new query only when the previous one has terminated. Without loss of generality, the response from a process to its own queries is assumed to always arrive among the first $(n-f)$ responses it is waiting for. Moreover, QUERY_ALIVE() and RESPONSE() are assumed to be implicitly tagged in order not to confuse RESPONSE() messages corresponding to different QUERY_ALIVE() messages.

Figure 1 depicts a query-response mechanism in a system made up of $n = 6$ processes, and $f = 2$. After p_3 broadcasts QUERY_ALIVE(), the $n - f = 4$ first RESPONSE() messages it receives are from p_2, p_3 (itself), p_5 and p_6. These responses are the winning responses for that query. Notice that p_4 has crashed.

2.2 Eventual Failure Detectors

Failure detectors have been formally defined by Chandra and Toueg [5]. Informally, a failure detector consists of a set of modules, each one attached to a process: the module attached to p_i maintains a set (named $suspected_i$) of processes it currently suspects to have crashed. Any failure detector module is inherently unreliable: it can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. Moreover, suspicions are not necessarily stable: a process p_j can be added to or removed

from a set $suspected_i$ according to whether p_i's failure detector module currently suspects p_j or not.

As indicated in the Introduction, a failure detector class is formally defined by two abstract properties, namely a *Completeness* property and an *Accuracy* property. The upper layer protocol that uses the corresponding failure detector relies on these properties to ensure its liveness. More precisely, the completeness property prevents a process from entering a deadlock, while the accuracy property prevents it from entering a livelock (either deadlock or livelock could prevent the process from terminating).

Chandra and Toueg have defined several classes of failure detectors [5]. In this paper, we are interested in two of them. They are defined from the following completeness property:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

and the following accuracy properties:

- Eventual Strong Accuracy: There is a time after which no correct process is suspected.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected.

Combining the completeness property with the previous accuracy properties provides the following classes of eventual failure detectors [5]:

- $\diamond\mathcal{P}$: The class of *Eventually Perfect* failure detectors. This class contains all the failure detectors that satisfy strong completeness and eventual strong accuracy.
- $\diamond\mathcal{S}$: The class of *Eventually Strong* failure detectors. This class contains all the failure detectors that satisfy strong completeness and eventual weak accuracy.

Clearly $\diamond\mathcal{P} \subset \diamond\mathcal{S}$. As already noticed in the Introduction, $\diamond\mathcal{S}$ is the weakest class of failure detectors for solving consensus in asynchronous systems prone to process crashes [6]. $\diamond\mathcal{P}$ is the weakest class of failure detectors for solving quiescent reliable communication in asynchronous systems with process crashes and lossy links [1][3].

2.3 Additional Assumptions

This section states two types of additional assumptions that allow the implementation of failure detectors in the class $\diamond\mathcal{S}$ or $\diamond\mathcal{P}$.

[3]The *quiescent reliable communication* problem consists in achieving reliable communication with protocols that eventually stop sending messages in an asynchronous distributed system where processes can crash and links can be lossy.

Partial synchrony The *partial synchrony* assumption has been stated in the Introduction [5, 7]. We recall it here:

- Property PS: There is a time t_S after which there are bounds on process speeds and message transfer delays (but those bounds are not known).

The intuition that underlies this property is the following. Generally, a system evolves through a sequence of long *stable* periods separated by (usually, relatively short) *unstable* periods. The stable periods are when "the behavior is predictable", i.e., when there are bounds on message delays and processing times. This is exactly what is captured by the PS property. The hope is then that each stable period will be "long enough" for the properties of the failure detector to be realized during that period. As "long enough" cannot be measured (remember that we are in an asynchronous system where there is no timing assumption), it is captured in the statement of the PS property by "there is a time t_S after which ..." (A similar clause will be used in the other properties stated below.)

Behavioral properties on the message exchange pattern We state here two behavioral properties. The first one has already been stated in the Introduction. Both are properties on the message exchange pattern involved in the query-response mechanism. (MP, w and s stand for "Message Pattern", "weak" and "strong", respectively.)

- Property MP_w (Weak Message Pattern): There are a time t_{Mw}, a correct process p_i and a set Q of $(f+1)$ processes (t_{Mw}, p_i and Q are not known in advance) such that, after t_{Mw}, each process $p_j \in Q$ gets a winning response from p_i to each of its queries (until p_j possibly crashes).
- Property MP_s (Strong Message Pattern): There is a time t_{Ms} after which, $\forall\ p_i$, there is a set Q_i of $(f+1)$ processes p_j such that any such p_j gets a winning response from p_i to each of its queries (until p_i or p_j crashes). As previously, the time t_{Ms} and the sets Q_i are not known in advance.

The intuition that underlies these properties is the following. Even if the system is not stable, it is possible that its behavior has some "regularity" that can be exploited to build failure detectors. This regularity can be seen as some "logical synchrony" (as opposed to "physical" synchrony captured by the property PS). More precisely, MP_w states that there is eventually a cluster Q of $(f+1)$ processes that (until some of them possibly crash) receive winning responses from p_i to their queries. This can be interpreted as

```
(1)  init: not_rec_from_i ← ∅; suspected_MP_i ← ∅; suspected_PS_i ← ∅;
(2)       for each j do Δ_i[j] ← default timeout value end do;

task T1:
     repeat periodically
(3)       foreach j do send QUERY_ALIVE() to p_j enddo;
(4)       wait until ( corresponding RESPONSE(not_rec_from) received from (n − f) processes );
(5)       let I = the set of processes from which p_i received a RESPONSE message;
(6)       let X = the set of the not_rec_from sets received from the processes in I;
(7)       not_rec_from_i ← Π − I;
(8)       suspected_MP_i ← ⋂_{x∈X} x
     end repeat

task T2:
(9)  upon reception of QUERY_ALIVE() from p_j
(10)      do send RESPONSE(not_rec_from_i) to p_j;
(11)         if (p_j ∈ suspected_PS_i) then Δ_i[j] ← Δ_i[j] + 1 end if;
(12)         suspected_PS_i ← {p_k | no QUERY_ALIVE() has been received
                                    from p_k during the last Δ_i[k] time units };
(13)      end do

task T3:
(14) when SUSPECT() is invoked by the upper layer
(15)      do let suspected_i = suspected_PS_i ∩ suspected_MP_i;
(16)         return (suspected_i)
(17)      end do
```

Figure 2. Failure Detector Module FD_i Associated with p_i

follows: among the n processes, there is a process that has $(f+1)$ "favorite neighbors" with which it communicates faster than with the other processes. When we consider the particular case $f = 1$, MP_w boils down to a simple channel property, namely, there is channel (p_i, p_j) that is never the slowest among the channels connecting p_j to the other processes (it is shown in [16] that the probability that this property be satisfied in practice is very close to 1). The intuition that underlies MP_s is similar.

Protocols implementing failure detectors belonging to $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ in asynchronous distributed systems satisfying the assumption PS are described in [5, 8, 13, 14]. A protocol implementing a failure detector belonging to $\diamond\mathcal{S}$ in asynchronous distributed systems satisfying MP_w is described in [16].

The next sections show how to build a failure detector of the class $\diamond\mathcal{S}$ (resp., $\diamond\mathcal{P}$) when the underlying system satisfies the property $MP_w \vee PS$ (resp., $MP_s \vee PS$). Let us notice that MP_w alone is too weak to build failure detector of the class $\diamond\mathcal{P}$.

3 A Hybrid Protocol for $\diamond\mathcal{S}$

3.1 Underlying Principles

The protocol borrows, combines and extends ideas from [5] and [16]. It is made of three tasks (Figure 2). The aim of the tasks $T1$ and $T2$ is to manage two sets of suspected processes, namely, $suspected\_MP_i$ (the set of processes suspected with respect to the assumption MP_w), and $suspected\_PS_i$ (the set of processes suspected with respect to the assumption PS). The aim of the task $T3$ is to answer the calls issued by the upper layer application when it invokes the primitive SUSPECT() (line 14). The caller is then provided with the set of process that are currently suspected, namely, $suspected\_MP_i \cap suspected\_PS_i$ (lines 15-16).

The other local variables managed by a process p_i are a set $not\_rec\_from_i$ and an array $\Delta_i[1..n]$. The set $not\_rec\_from_i$ is used to keep the processes whose responses to the last query of p_i have not been received among the $(n-f)$ first ones. $\Delta_i[j]$ is a timer used by p_i to detect the possible crash of p_j when considering the assumption PS.

Periodically, p_i issues QUERY_ALIVE() messages (line 3). Let us notice that the repetition period of task $T1$ has no incidence on the correctness of the protocol. (It does of course influence the latency in a practical system.) A QUERY_ALIVE() message has a double aim: activation of a query-response mechanism (this is related to the MP_w assumption), and indication to the other processes that p_i is alive (this is related to the PS assumption). After it has issued a query, p_i waits until it has received $(n-f)$ corresponding responses (line 4). Then it determines the set of f processes from which it has not received responses and

updates $not\_rec\_from_i$ accordingly (line 7).

The management of the set $suspected\_MP_i$ is more subtle. Let us observe that each RESPONSE() message carries the current value of the set $not\_rec\_from_j$ of its sender p_j (line 10). After having received the first $(n - f)$ responses it was waiting for, p_i updates $suspected\_MP_i$ to be equal to the the intersection of the $(n - f)$ $not\_rec\_from_j$ sets it considers (line 8). This means that $suspected\_MP_i$ consists of the processes that have not sent a RESPONSE() message to the last query of p_i (because they have crashed) or whose RESPONSE() messages were losing messages for that query. As we will show in the proof, if the system satisfies the property MP_w, the sets $suspected\_MP_i$ satisfy the properties defining $\diamond\mathcal{S}$.

Let us now consider the management of the set $suspected\_PS_i$. Each time a process p_i receives a QUERY_ALIVE() message from some process p_j, it increments the timeout value associated with p_j if $p_j \in suspected\_PS_i$ (line 11). Then, p_i recomputes the value of this set which, from now on, includes the processes p_k from which p_i has not received a QUERY_ALIVE() message since $\Delta_i[k]$ time units (the timeout value has been exceeded without receiving a QUERY_ALIVE() message). Let us observe that the recomputed value of $suspected\_MP_i$ at line 12 will not include p_j since p_i just received a QUERY_ALIVE() message from p_j and $\Delta_i[j] > 0$.

As we will show in the proof, if the system satisfies the property PS, the sets $suspected\_PS_i$ satisfy the properties defining $\diamond\mathcal{S}$.

3.2 Proof of the Protocol

Let us first observe that, as there are at most f ($1 \leq f < n$) processes that can crash and channels do not lose messages, no FD_i module can block forever. Moreover, we assume by definition that the $not\_rec\_from_i$ set of a crashed process p_i is equal to $\emptyset$. (This definition is motivated by the fact that a crashed process never sends its $not\_rec\_from_i$ set to the other processes.) The proof of Lemma 1 and Lemma 4 are from [5].

Lemma 1 *Every process that crashes eventually belongs permanently to the set $suspected\_PS_i$ of every correct process p_i.*

Proof Let p_j be a process that crashes at time t. Let us first observe that p_j does not send QUERY_ALIVE() messages after t. Moreover, as message delays are finite, there is a time t'' after which all the QUERY_ALIVE() messages sent by p_j before t have been received. Moreover, p_i receives QUERY_ALIVE() messages after $t' = t'' + \Delta_i[j]$ (as p_i has not crashed, it receives at least its own QUERY_ALIVE() messages). Each time it receives such a message after t', it includes p_j in the set $suspected\_PS_i$ (line 12). Hence, p_j remains permanently in this set. $\Box_{Lemma\ 1}$

Lemma 2 *Every process that crashes eventually belongs permanently to the set $suspected\_MP_i$ of every correct process p_i.*

Proof After a process p_j has crashed, it no longer responds to the QUERY_ALIVE() messages it receives. It follows that eventually p_j belongs permanently to the $not\_rec\_from_i$ sets of each process that does not crash. Hence, p_j belongs to the intersection of all these sets received by a correct process p_i, and consequently, p_j becomes a member of the set $suspected\_MP_i$. $\Box_{Lemma\ 2}$

Lemma 3 [Strong Completeness] *The protocol described in Figure 2 satisfies the strong completeness property (i.e., every process that crashes is eventually suspected by every correct process).*

Proof The proof follows directly from the Lemmas 1 and 2 and the fact that the set $suspected_i$ is defined as the intersection of $suspected\_PS_i$ and $suspected\_MP_i$: as both sets eventually contain all crashed processes, so does their intersection. $\Box_{Lemma\ 3}$

Let us observe that the previous lemmas do not involve any additional assumption to the asynchronous system model. This is not the case for the two lemmas that follow.

Lemma 4 *Let us assume that the underlying system satisfies the property PS. Then, there is a time t such that, after t, for each pair of correct processes p_i and p_k, we have $p_i \notin suspected\_PS_k$.*

Proof We show that, when the system satisfies the property PS, there is a time after which the set $suspected\_PS_i$ of each correct process p_i includes no correct process. Let p_j be a correct process. It sends periodically QUERY_ALIVE() messages. Due to the property PS (eventual existence of bounds on message delays and processing time), and the incrementation of the timeout value $\Delta_i[j]$, should process p_j be suspected (line 11), there is a time t after which the timeout value $\Delta_i[j]$ is greater than or equal to the maximum duration that elapses between any two consecutive receptions by p_i of QUERY_ALIVE() messages sent by p_j. It then follows that if p_j was in $suspected\_PS_i$ before t, it is removed from it (line 11), and is never added again to this set in the future (line 12). Hence, p_j never belongs to the set $suspected\_PS_i$ of correct process p_i after t. $\Box_{Lemma\ 4}$

Lemma 5 *Let us assume that the underlying system satisfies the property MP_w. Then, there is a time t and a correct process p_i such that, after t and for each correct process p_k, we have $p_i \notin suspected\_MP_k$.*

Proof *Claim*: If MP_w holds, then there is a time t and a correct process p_i that does not belong to the $not\_rec\_from_j$ sets of at least $(f+1)$ processes (some of those processes can be crashed).
Proof of the claim. As MP_w holds, there is a time t after which there are a correct process p_i and a set Q of $(f+1)$ processes such that $\forall p_j \in Q$: p_j gets a response from p_i to each of its queries (until p_j possibly crashes).

Let $p_j \in Q$. If p_j has crashed at a time $\leq t$, then by definition we have $p_i \notin not\_rec\_from_j$. Otherwise, we conclude from MP_w that (until p_j possibly crashes) the response from p_i to each of its queries is a winning response. So, each time it executes line 5, p_j puts p_i in the set I. It follows that each time p_j executes line 7, we have $p_i \notin not\_rec\_from_j$. *End of the proof of the claim.*

We now show that there is a time t and a correct process p_i such that the set $suspected\_MP_k$ of any correct process p_k never includes p_i after t.

As MP_w is satisfied, it follows from the claim that there is a correct process p_i that, after some finite time t', never belongs to the $not\_rec\_from_j$ sets of at least $(f+1)$ processes. Let $t \geq t'$ be a time after which all RESPONSE() messages sent before t by the processes of Q have been received (or discarded).

After t, p_i can appear in the RESPONSE() messages of at most $(n-(f+1))$ processes, from which we conclude that, for any correct process p_k, there is at least one RESPONSE(not_rec_from) message that does not carry the identity of p_i, when it executes line 8. It follows that after t, p_i can no longer appear in the set $suspected\_MP_k$ of a correct process p_k after t. $\square_{Lemma\ 5}$

Theorem 1 *Let $1 \leq f < n$. The protocol described in Figure 2 implements a failure detector of the class $\diamond\mathcal{S}$ when the underlying system satisfies the property MP_w or the property PS.*

Proof The proof follows directly from Lemmas 3, 4 and 5 and the fact that the set $suspected_i$ is defined as being equal to $suspected\_PS_i \cap suspected\_MP_i$. $\square_{Theorem\ 1}$

4 The Case of $\diamond\mathcal{P}$

Interestingly, the previous protocol implements a failure detector of the class $\diamond\mathcal{P}$ when the underlying system satisfies one of the properties MP_s or PS (as already noted, MP_w alone is too weak to build a failure detector of the class $\diamond\mathcal{P}$ while PS alone is strong enough). When we consider MP_s, the protocol additionally requires $f < n/2$. This is a feasibility condition motivated by the following reason. Let us consider the case where there is no failure. In order that MP_s might be satisfied we need:

- $\forall\, p_i$: $(f+1)$ processes have to receive winning RESPONSE() messages from p_i to their queries. This means that $n(f+1)$ receptions of winning RESPONSE() messages have to be possible (i.e., not discarded).

- $\forall\, p_i$: a process accepts only $(n-f)$ RESPONSE() messages to each of its queries (as losing responses are discarded). This means that at most $n(n-f)$ receptions are possible.

Combining the two previous observations, we get $n(n-f) \geq n(f+1)$, i.e., $n > 2f$. So, for MP_s to be possibly satisfied in an asynchronous system, a majority of processes has to be correct.

Assuming $f < n/2$, we now show that the protocol described in Figure 2 implements a failure detector of the class $\diamond\mathcal{P}$ when the underlying system satisfies one of the properties MP_s or PS.

Lemma 6 [Eventual Strong Accuracy] *Let us assume that the underlying system satisfies the property PS or MP_s. Then, the protocol described in Figure 2 guarantees that eventually no correct process is suspected by the correct processes.*

Proof (Sketch)
Considering any correct process p_i, and its sets $suspected\_PS_i$ and $suspected\_MP_i$, we have:

- Due to Lemma 1, the set $suspected\_PS_i$ eventually contains all crashed processes.

- Due to Lemma 2, the set $suspected\_MP_i$ eventually contains all crashed processes.

- When PS is satisfied, it follows from Lemma 4 that there is a time after which $suspected\_PS_i$ contains no correct process.

- When MP_s is satisfied, it can similarly be shown that there is a time after which the set $suspected\_MP_i$ contains no correct process (the proof is similar to the proof of Lemma 5, considering MP_s instead of MP_w).

It follows from these observations that, after some time, the set $suspected_i$ (i.e., $suspected\_PS_i \cap suspected\_MP_i$) always includes all crashed processes plus possibly some correct processes if the system satisfies neither PS nor MP_s. We then conclude that, as soon as PS or MP_s is satisfied, the set $suspected_i$ of each correct process p_i satisfies the eventual strong accuracy property. $\square_{Lemma\ 6}$

Theorem 2 *Let $f < n/2$. The protocol described in Figure 2 implements a failure of the class $\diamond\mathcal{P}$ when the underlying system satisfies the property MP_s or the property PS.*

Proof The proof follows directly from Lemmas 3 and 6. $\square_{Theorem\ 2}$

It follows that the proposed hybrid protocol has a generic dimension, namely, without any modification, it provides us with a failure detector that belongs to the class:

- $\diamond\mathcal{S}$ if the underlying distributed system satisfies $\big(MP_w \wedge f < n\big) \vee PS$.
- $\diamond\mathcal{P}$ if the underlying distributed system satisfies $\big(MP_s \wedge f < n/2\big) \vee PS$.

5 Conclusion

This paper has shown that, when one is interested in implementing failure detectors with eventual accuracy properties, approaches based on the partial synchrony assumption or on the statement of behavioral properties on the message exchange pattern, are not antagonistic and can even be advantageously combined. A hybrid protocol, based on such a combination and implementing eventual failure detectors has been presented. Interestingly, this protocol benefits from the best of both worlds in the sense that it converges (i.e., provides a failure detector with the required properties) as soon as either the system behaves synchronously or the required message exchange pattern is satisfied. This shows that considering that the underlying system can satisfy several alternative assumptions can help expedite convergence at no additional cost. Since convergence is guaranteed if any one of the alternative assumptions is satisfied, the proposed hybrid approach also provides increased overall assumption coverage for free [19].

References

[1] Aguilera M.K., Chen W. and Toueg S., On Quiescent Reliable Communication. *SIAM Journal of Computing*, 29(6):2040-2073, 2000.

[2] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Stable Leader Election. *Proc. 15th Symposium on Distributed Computing (DISC'01)*, Lisbon (Portugal), Springer Verlag LNCS #2180, pp. 108-122, 2001.

[3] Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

[4] Bertier M., Marin O. and Sens P., Implementation and Performance Evaluation of an Adaptable Failure Detector. *Proc. Int. IEEE Conferecne on Dependable Systems and Networks (DSN'02)*, IEEE Computer Society Press, pp. 354-363, Washington D.C., 2002.

[5] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[6] Chandra T.D., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[7] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.

[8] Fetzer C., Raynal M. and Tronel F., An Adaptive Failure Detection Protocol. *Proc. 8th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'01)*, IEEE Computer Society Press, pp. 146-153, Seoul (Korea), 2001.

[9] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[10] Gupta I., Chandra T.D. and Goldszmidt G.S., On Scalable and Efficient Distributed Failure Detectors. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press, pp. 170-179, Newport (RI), 2001.

[11] Hélary J.-M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Process Crashes. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):897-909, 2000.

[12] Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395-408, 2002.

[13] Larrea M., Arèvalo S. and Fernández A., Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. *Proc. 13th Symposium on Distributed Computing (DISC'99)*, Bratislava (Slovakia), Springer Verlag LNCS #1693, pp. 34-48, 1999.

[14] Larrea M., Fernández A. and Arèvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Society Press, pp. 52-60, Nuremberg (Germany), 2000.

[15] Mostefaoui A., Mourgaya E. and Raynal M., An Introduction to Oracles for Asynchronous Distributed Systems. *Future Generation Computer Systems*, 18(6):757-767, 2002.

[16] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous Implementation of Failure Detectors. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360, San Francisco (CA), 2003.

[17] Mostefaoui A., Raynal M. and Tronel F., The Best of Both Worlds: a Hybrid Approach to Solve Consensus. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'00)*, IEEE Computer Society Press, pp. 513-522, New-York City, June 2000.

[18] Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228-234, 1980.

[19] Powell D., Failure Mode Assumptions and Assumption Coverage. *Proc. of the 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, Boston, MA, pp.386-395, 1992.

[20] Raynal M., A Short introduction to Failure Detectors for Asynchronous Distributed Systems. Submitted to publication, *Tech Report*, IRISA, Université de Rennes 1 (France), November 2003.

[21] Raynal M. and Tronel F., Group Membership Failure Detection: a Simple Protocol and its Probabilistic Analysis. *Distributed Systems Engineering Journal*, 6(3):95-102, 1999.