Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 361–373
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Eventually consistent failure detectors☆

Mikel Larrea[a,*], Antonio Fernández[b], Sergio Arévalo[b]

[a] *Departamento de Arquitectura y Tecnología de Computadores, Universidad del País Vasco, Paseo Manuel de Lardizabal 1, 20018 San Sebastián, Spain*
[b] *Universidad Rey Juan Carlos, 28933 Móstoles, Spain*

Received 26 December 2001; received in revised form 1 October 2004
Available online 22 January 2005

Abstract

The concept of *unreliable failure detector* was introduced by Chandra and Toueg as a mechanism that provides information about process failures. This mechanism has been used to solve different problems in asynchronous systems, in particular the Consensus problem. In this paper, we present a new class of unreliable failure detectors, which we call *Eventually Consistent* and denote by $\diamond\mathcal{C}$. This class combines the failure detection capabilities of class $\diamond\mathcal{S}$ with the eventual leader election capability of class Ω. This capability allows all correct processes to eventually choose the same correct process as *leader*. We study the relationship between $\diamond\mathcal{C}$ and other classes of failure detectors. We also propose an efficient algorithm to transform $\diamond\mathcal{C}$ into $\diamond\mathcal{P}$ in models of partial synchrony. Finally, to show the power of this new class of failure detectors, we present a Consensus algorithm based on $\diamond\mathcal{C}$. This algorithm successfully exploits both the leader election and the failure detection capabilities of the failure detector, and performs better in number of rounds than all the previously proposed algorithms for $\diamond\mathcal{S}$.

Keywords: Consensus problem; Crash failures; Distributed systems; Failure detection; Leader election; Partial synchrony; Unreliable failure detectors

1. Introduction

1.1. Unreliable failure detectors

An *unreliable failure detector* is a mechanism that provides (possibly incorrect) information about faulty processes. As originally defined, when queried by a process, a failure detector returns a set of processes believed to have crashed (suspected processes). The sets returned to different processes can be different, and can contain processes that have not crashed. The concept of unreliable failure detector was introduced by Chandra and Toueg in [6], where they proposed several classes of unreliable failure detectors. After this seminal work, a number of authors have proposed other classes of unreliable failure detectors [1,4,11,19]. These failure detectors have been used to solve several fundamental problems in asynchronous distributed systems (e.g., consensus [6]).

In [6], failure detectors were characterized in terms of two properties: *completeness* and *accuracy*. Completeness characterizes the failure detector capability of suspecting every incorrect process (processes that actually crash), while accuracy characterizes the failure detector capability of not suspecting correct processes. Two kinds of completeness and four kinds of accuracy were defined in [6], which combined yield eight classes of failure detectors.

In this paper, we focus on the following completeness and accuracy properties, from those defined in [6]. We say that a process p *suspects* a process q if q is in the set of suspected processes returned by the failure detector when queried by p.

☆ Research partially supported by the Spanish Research Council, under Grants TIC99-0280-C02-02, TEL99-0582, TIC98-1032-C03-01, and TIC2001-1586-C03-01, the Comunidad de Madrid under Grant 07T/0022/2003, and the Universidad Rey Juan Carlos under Grant PPR-2003-37.

* Corresponding author. Fax: +34 943 015590.
E-mail addresses: mikel.larrea@si.ehu.es (M. Larrea), afernandez@acm.org (A. Fernández), s.arevalo@escet.urjc.es (S. Arévalo).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.11.001

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

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

- *Strong completeness*: Eventually every process that crashes is permanently suspected by *every* correct process.
- *Weak completeness*: Eventually every process that crashes is permanently suspected by *some* correct process.
- *Eventual strong accuracy*: There is a time after which correct processes are not suspected by any correct process.
- *Eventual weak accuracy*: There is a time after which some correct process is never suspected by any correct process.

Combining in pairs these completeness and accuracy properties, we obtain four different failure detector classes, which are shown in Fig. 1. In [6], Chandra and Toueg also proposed an algorithm to implement a $\diamond\mathcal{P}$ failure detector in models of partial synchrony. Other more efficient algorithms under similar models have been proposed to implement failure detectors of all these classes [15], and specifically of class $\diamond\mathcal{S}$ [16].

In [5] Chandra et al. defined another class of failure detectors, denoted Ω, and used it to prove that $\diamond\mathcal{W}$ is the weakest failure detector class for solving Consensus. [1] When queried by a process p, a failure detector of class Ω returns a *single* process q, currently considered to be *correct* (we say that p *trusts* q). A failure detector in Ω satisfies the following property:

Property 1. *There is a time after which every correct process permanently trusts the same correct process.*

This property of Ω failure detectors can be seen as an *eventual leader election capability*, since it guarantees that, eventually, all correct processes will agree on trusting the same correct process (a leader). This capability does not give knowledge of when the leader has been elected and allows the existence of several trusted processes at the same time. Note that, translating the concept of trusted processes to the classical of suspected processes, implicitly Ω failure detectors always suspect all processes except one. This implies that, in general, Ω failure detectors provide less accuracy (information about correct processes) than detectors from the classes in Fig. 1.

There are several algorithms implementing Ω failure detectors in the literature. Chandra et al. [5] and Chu [7] show how to transform a $\diamond\mathcal{W}$ failure detector into an Ω failure detector in an asynchronous system. The algorithm to implement $\diamond\mathcal{S}$ proposed in [16] also implicitly implements an Ω failure detector. More recently, Aguilera et al. [2] have proposed several algorithms for implementing Ω that are *stable*, i.e., once a leader is elected, it remains the leader for as long as it does not crash and its links behave well. Finally, Aguilera et al. [3] have presented systems with weak requirements in which Ω failure detectors can be implemented, while $\diamond\mathcal{P}$ failure detectors cannot.

[1] To prove their result Chandra et al. show first that Ω is at least as strong as $\diamond\mathcal{W}$, and then that any failure detector $\mathcal{D}$ that can be used to solve Consensus is at least as strong as Ω (and hence at least as strong as $\diamond\mathcal{W}$).

1.2. Consensus algorithms

The Consensus problem [21] is a fundamental problem in distributed systems. It was shown by Fischer et al. [9] that it is impossible to solve Consensus deterministically in a pure asynchronous system. Chandra and Toueg showed in [6] that their unreliable failure detectors allow to solve Consensus in asynchronous systems. Since then, several distributed fault-tolerant algorithms to solve Consensus based on unreliable failure detectors have been proposed [12,18,22].

Most Consensus algorithms based on failure detectors with eventual accuracy require at least a failure detector of class $\diamond\mathcal{S}$ and proceed in rounds. In each round a different process acts as coordinator, following a prearranged sequence. This approach is known as the *rotating coordinator paradigm*. If the coordinator of a round crashes or is suspected by several processes, the round may fail and the consensus is not reached in that round. With a failure detector of class $\diamond\mathcal{S}$ it is guaranteed the existence of a correct process, namely *leader*, that is eventually not suspected by any correct process. If after no process suspects it, *leader* eventually becomes the coordinator, the consensus is guaranteed to be reached.

The inconvenience of the $\diamond\mathcal{S}$-Consensus algorithms based on the rotating coordinator paradigm is that, if *leader* does not become coordinator until i rounds after no process suspects it, the processes may have to wait these i rounds to reach consensus (where i could be $\Omega(n)$, with n being the number of processes). It would be nice to have Consensus algorithms that quickly choose *leader* as the coordinator, hence reducing the number of rounds required to reach consensus.

There have been other approaches to solve the Consensus problem in non-synchronous systems. In [8], Dwork et al. assumed partially synchronous models and proposed Consensus algorithms for these models. These algorithms also use

the rotating coordinator paradigm and can present the above problem.

To our knowledge, the first Consensus algorithm that uses a kind of eventual leader election algorithm to choose the coordinator of a round (instead of using a rotating coordinator approach) is the Paxos Consensus algorithm [13]. In this algorithm a system model different from the above is used, in which periods of synchrony and asynchrony alternate. The Paxos Consensus algorithm proceeds in asynchronous rounds, with a coordinator, given by the leader election algorithm, for each round.

Recently, Mostefaoui and Raynal [20] have proposed a Consensus algorithm, based on an Ω failure detector, that does not use the rotating coordinator paradigm. Hence, their algorithm does not present the above problem. However, instead it has to deal with the lack of accuracy of the Ω failure detectors.

1.3. Our contributions

In this paper, we define a new class of unreliable failure detectors that combines the completeness and accuracy of the $\diamond\mathcal{S}$ failure detectors and the eventual leader election capability of Ω failure detectors. We call this class *Eventually Consistent* and denote it by $\diamond\mathcal{C}$. The main property of the failure detectors in this class is that they provide simultaneously the classical failure detection functionality (i.e., a set of suspected processes) of $\diamond\mathcal{S}$ failure detectors and the eventual leader election capability (i.e., a common trusted process) of Ω failure detectors.

Considering the Consensus problem, the eventual leader election functionality of the $\diamond\mathcal{C}$ failure detectors allows every correct process to eventually agree on a coordinator that can be used to reach consensus. Hence, with these failure detectors we do not need to rely on the rotating coordinator paradigm for eventually choosing an appropriate coordinator. Additionally, the consensus algorithm may take advantage of the accuracy and completeness properties of the $\diamond\mathcal{C}$ failure detectors to speed up the agreement. We present a Consensus algorithm that uses these properties.

We study the relationship between $\diamond\mathcal{C}$ and the classes of failure detectors presented in Fig. 1. We first observe that any implementation of $\diamond\mathcal{P}$ can be trivially used to implement $\diamond\mathcal{C}$. Similarly, we observe that $\diamond\mathcal{C}$ can be implemented on top of any implementation of $\diamond\mathcal{S}$ (and hence of $\diamond\mathcal{W}$) by means of an asynchronous distributed algorithm [5,7]. We then show that $\diamond\mathcal{C}$ can be implemented as efficiently as $\diamond\mathcal{S}$ in models of partial synchrony [6,8]. To show that, we observe that the $\diamond\mathcal{S}$ failure detectors implemented by the efficient algorithms presented in [15,16] can be used to implement $\diamond\mathcal{C}$ at no additional cost in terms of message exchanges.

Then, we propose an efficient algorithm to transform any failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$ into a failure detector of class $\diamond\mathcal{P}$ in a model of partial synchrony. The transformation algorithm only requires the input links to the leader chosen by $\mathcal{D}$ to be partially synchronous (eventually there is an unknown bound on the delay suffered by messages) and the output links to be fair (messages can be lost, but if infinite messages are sent, then infinite messages are received). Eventually only these links carry messages. Recently, Aguilera et al. [2] have proposed an algorithm following a similar approach for implementing $\diamond\mathcal{P}$ based on an Ω failure detector. This algorithm assumes a weak model of partial synchrony, in which only n bidirectional links are required to be eventually timely. In their algorithm eventually only these bidirectional links carry messages.

Finally, to show the power of this class of failure detectors, we present a Consensus algorithm based on $\diamond\mathcal{C}$. This algorithm proceeds in asynchronous rounds and each round is divided in several phases, like most previous $\diamond\mathcal{S}$-Consensus algorithms [6,12,18,22]. The main difference of our algorithm is the way the coordinator is selected. We do not use the rotating coordinator paradigm as they do, but the leader election capability of $\diamond\mathcal{C}$. This algorithm reaches consensus in at most one round after the leader election property of $\diamond\mathcal{C}$ is satisfied, while we show that any rotating coordinator $\diamond\mathcal{S}$-Consensus algorithm requires $n-1$ additional rounds after stabilization in the worst case.

In this Consensus algorithm we introduce an additional improvement that makes use of the accuracy and completeness properties of $\diamond\mathcal{C}$. In previous algorithms (see for instance the $\diamond\mathcal{S}$-Consensus algorithm in [6]), the round coordinator proposes a value and waits for replies (accepting or not that value) from a majority of processes (the existence of a majority of correct processes is a requirement). If any of these replies is negative, the decision is not made. One single negative reply blocks the decision. In our algorithm, the coordinator waits for replies as long as a decision can still be made in the round. Then, if a majority of replies are positive, the decision is made, even if there are negative replies.

Compared with the Paxos Consensus algorithm [13] mentioned above, our algorithm works in an asynchronous system extended with a failure detector, while Paxos assumes a system model in which there are periods of synchrony. Apart from that, the leader election algorithm proposed in [13] is basically a failure detection algorithm. However, it strongly relies on the existence of long enough periods of synchrony in the system, as the consensus algorithm itself does. Besides that, both algorithms use similar approaches.

Recently, Mostefaoui and Raynal [20] have proposed a Consensus algorithm, based on an Ω failure detector, that does not use the rotating coordinator paradigm. This work is inspired on previous versions of the present paper [14,17]. This algorithm has to deal with the lack of accuracy of the Ω detector. Since the detector only gives information about one process, in order to make a decision, the coordinator does not have any information about which processes may reply. To prevent one single negative reply from blocking the decision as we mentioned above, the coordinator waits for $n-f$

replies, where f is an upper bound on the number of processes that can fail. The number $n - f$ can be much smaller than the number of replies that could in fact be received, and a small number of negative replies can block the decision. In fact, if all it is known is that there is a majority of correct processes (i.e., $f < n/2$), the algorithm only waits for a majority of replies as above, and one negative reply blocks the decision.

The rest of the paper is organized as follows. In Section 2, we establish the model of the system we use in the rest of the paper and define the new class of failure detectors ($\diamond\mathcal{C}$). In Section 3, we study its relationship with other classes of failure detectors. In Section 4, we propose an efficient algorithm to transform $\diamond\mathcal{C}$ into $\diamond\mathcal{P}$ in models of partial synchrony. In Section 5, we present an efficient Consensus algorithm based on $\diamond\mathcal{C}$. Finally, Section 6 concludes the paper.

2. Definitions

2.1. System model

We consider a distributed system consisting of a finite totally ordered set Π of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$. Processes communicate only by sending and receiving messages. Every pair of processes is assumed to be connected by two reliable communication links (in opposite directions). Unless stated otherwise, the system is *asynchronous*, i.e., there are no timing assumptions about neither the relative speeds of the processes nor the delay of messages. Processes can fail by *crashing*, that is, by prematurely halting. Crashes are permanent, i.e., crashed processes do not recover.

A *distributed failure detector* can be viewed as a set of n failure detection modules, each one attached to a different process in the system. These modules cooperate to satisfy the required properties of the failure detector. A process requests information about failures to its attached failure detector module by requesting a set of suspected processes (e.g., to a $\diamond\mathcal{S}$ failure detector) or by requesting the identity of a trusted process (e.g., to an Ω failure detector). We will denote by $\mathcal{D}.suspected_p$ the set of suspected processes returned by a failure detector $\mathcal{D}$ to a given process p. Similarly, we will denote by $\mathcal{D}.trusted_p$ the trusted process returned by a failure detector $\mathcal{D}$ to a process p. These suspected and trusted processes can differ from one process to another at a given time. We assume that a process interacts only with its local failure detection module.

2.2. Eventually consistent failure detectors

We introduce now the class of eventually consistent failure detectors $\diamond\mathcal{C}$. The main characteristic of these failure detectors is that they combine the characteristics of $\diamond\mathcal{S}$ and Ω failure detectors. Then, a failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$ attends two types of requests from the processes in the system, requests for sets of suspected processes and requests for trusted processes. The sets of suspected processes satisfy the conditions of a $\diamond\mathcal{S}$ failure detector and the trusted processes satisfy the conditions of an Ω failure detector.

Definition 1. A failure detector belongs to the *Eventually Consistent* class of failure detectors, denoted $\diamond\mathcal{C}$, if it provides to every process p with a set of suspected processes $\mathcal{D}.suspected_p$ and one trusted process $\mathcal{D}.trusted_p$, such that,

- the sets $\mathcal{D}.suspected_p$ satisfy strong completeness and eventual weak accuracy (like $\diamond\mathcal{S}$),
- the trusted processes $\mathcal{D}.trusted_p$ satisfy Property 1 (like Ω), and
- there is a time after which the trusted processes are not suspected, i.e., $\mathcal{D}.trusted_p \notin \mathcal{D}.suspected_p$.

Observe that the completeness and accuracy properties of the sets $\mathcal{D}.suspected_p$ are the same as those of $\diamond\mathcal{S}$. Hence a failure detector of class $\diamond\mathcal{C}$ can be seen as a $\diamond\mathcal{S}$ failure detector enhanced with an eventual leader election mechanism (provided by the trusted processes $\mathcal{D}.trusted_p$). This mechanism guarantees that after some point in time all correct processes converge to a leader process. (However, it does not provide any knowledge of when the leader has been elected, and allows the existence of several leaders at the same time.) This property can be used by algorithms in which the safety properties are not affected by the simultaneous existence of several leaders, and that guarantee termination if a unique leader exists. As it is well known, usually it is not necessary for the failure detector to reach permanent stability to be useful. Instead, many algorithms can successfully complete if the failure detector provides a unique leader for long enough periods of time. Furthermore, these failure detectors can be very useful to algorithms that have early termination when there is a unique leader.

Finally, note that this definition of $\diamond\mathcal{C}$ does not impose that all processes but one must be suspected, as the definition of Ω alone implicitly does. This means that $\diamond\mathcal{C}$ can have a higher degree of accuracy than Ω.

3. Relation between $\diamond\mathcal{C}$ and other failure detector classes

In this section we will briefly study the relationship between the class $\diamond\mathcal{C}$ and other classes of failure detectors. First, observe that a $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ can be trivially built on top of any Ω failure detector $\mathcal{D}_\Omega$. $\mathcal{D}$ simply returns to a process p as the trusted process $\mathcal{D}.trusted_p$ the process $\mathcal{D}_\Omega.trusted_p$ returned by $\mathcal{D}_\Omega$, and as the set of suspected processes $\mathcal{D}.suspected_p$ all the processes except this trusted process. This transformation is very simple and efficient (no extra messages are needed). However, it offers very poor accuracy.

Observe also that any $\diamond\mathcal{P}$ failure detector $\mathcal{D}_{\diamond\mathcal{P}}$ can be used to implement a $\diamond\mathcal{C}$ failure detector $\mathcal{D}$. With $\mathcal{D}_{\diamond\mathcal{P}}$,

eventually the set of processes suspected by every correct process becomes the same, containing only all the processes that actually crash. Then $\mathcal{D}$ returns to a process p as the set of suspected processes $\mathcal{D}.suspected_p$ the set $\mathcal{D}_{\diamond\mathcal{P}}.suspected_p$ returned by $\mathcal{D}_{\diamond\mathcal{P}}$, and as the trusted process $\mathcal{D}.trusted_p$ the first (with respect to the order $p_1, \ldots, p_n$ assumed in the system model) process not in that set.

Similarly, a $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ can be implemented on top of any failure detector in classes $\diamond\mathcal{W}$ or $\diamond\mathcal{S}$ as follows. First, we have that an Ω failure detector $\mathcal{D}_\Omega$ can be obtained from any $\diamond\mathcal{W}$ (and hence $\diamond\mathcal{S}$) failure detector [5,7]. Then, $\mathcal{D}$ returns to a process p as the trusted process $\mathcal{D}.trusted_p$ the process $\mathcal{D}_\Omega.trusted_p$ returned by $\mathcal{D}_\Omega$. If the original detector is a $\diamond\mathcal{W}$ failure detector, it is transformed into a $\diamond\mathcal{S}$ failure detector [6]. As a consequence we have a $\diamond\mathcal{S}$ failure detector $\mathcal{D}_{\diamond\mathcal{S}}$, and $\mathcal{D}$ returns to a process p as the set of suspected processes $\mathcal{D}.suspected_p$ the set $\mathcal{D}_{\diamond\mathcal{S}}.suspected_p$ returned by that detector.

However, the transformation protocols from $\diamond\mathcal{W}$ to Ω of Chandra et al. [5] and Chu [7] are expensive in the number of messages exchanged, since they require that every process send messages periodically to all processes in the system. Fortunately, there are $\diamond\mathcal{S}$ failure detectors that can be used to build a $\diamond\mathcal{C}$ failure detector at no additional cost. An example is a $\diamond\mathcal{S}$ failure detector that guarantees that the first correct process (in some known order) is eventually not suspected by any correct process. Then, using a similar technique as the one above for $\diamond\mathcal{P}$ failure detectors, we can obtain a $\diamond\mathcal{C}$ failure detector without extra cost. One algorithm that satisfies these requirements is the $\diamond\mathcal{S}$ ring-based algorithm proposed in [15] for a partially synchronous model. With this algorithm, the set of non-suspected processes can be different in different processes, but the algorithm guarantees that eventually the first (starting from a special process *initial candidate to leader* and following the order defined by the ring) non-suspected process is the same for every correct process, and that it is correct.

4. Transforming $\diamond\mathcal{C}$ into $\diamond\mathcal{P}$ in models of partial synchrony

In this section, we present an efficient algorithm that transforms any failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$ into a failure detector $\mathcal{D}_{\diamond\mathcal{P}}$ of class $\diamond\mathcal{P}$. The approach followed is to use the eventually agreed trusted process p_{leader} provided by $\mathcal{D}$ to build and propagate a list of suspected processes that satisfies the properties of $\diamond\mathcal{P}$.

The transformation algorithm requires that the $n-1$ input links of p_{leader} are reliable and follow a model of partial synchrony like those considered in [6,8]. In this model it is assumed that after some finite global stabilization time *GST*, every message sent is received and processed in at most a bounded but unknown time Δ. The algorithm also requires the $n-1$ output links of p_{leader} to be fair, which means that they can lose messages, but if an infinite number of messages is sent, then an infinite number of the messages sent (and only those) is received (if the destination is correct). There are no restrictions on the rest of links.

Fig. 2 presents the algorithm in detail, which works as follows. Each *leader* process (i.e., each process p that considers itself as leader because $\mathcal{D}.trusted_p = p$) builds a local list of suspected processes by using time-outs (Tasks 3 and 4), and sends its list periodically to the rest of processes (Task 1). Concurrently, each process periodically sends an I-AM-ALIVE message to its trusted process (Task 2). Finally, when a process receives a list of suspected processes from its trusted process, it adopts this list as its own list (Task 5). Note that the algorithm only uses detector $\mathcal{D}$ to query for its trusted process. Hence, this algorithm could also be used to transform an Ω failure detector into a $\diamond\mathcal{P}$ failure detector.

Theorem 1. *Given a failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$ and a partially synchronous system as described, the algorithm of Fig. 2 implements a failure detector of class $\diamond\mathcal{P}$.*

Proof. The proof relies on the property satisfied by $\mathcal{D}.trusted_p$: there is a time t after which all the correct processes permanently *trust* the same correct process p_{leader}. Let us assume the system has become stable, i.e., we have reached a time $t' > \max(t, GST)$. Then,

(1) by Task 3, eventually every process that crashes is permanently suspected by p_{leader}, and

(2) by Tasks 2, 3, 4, and 5 there is a time after which correct processes are not suspected by p_{leader}. This can be easily shown. First note that p_{leader} never suspects itself (in Tasks 3 and 5). Then, for the rest of processes we use contradiction. Assume the period of Task 2 in all processes is Φ. Then, once p_{leader} suspects a correct process q, it receives an I-AM-ALIVE message from q in at most $2\Phi+\Delta$ time, by Task 2 and the fact that the link from q to p_{leader} is partially synchronous. From Task 4, p_{leader} stops suspecting q and increases the time-out interval $\Delta_p(q)$. Let us suppose by way of contradiction that q is suspected by p_{leader} an infinite number of times. After a bounded number of times the time-out $\Delta_p(q)$ will be larger than $2\Phi+\Delta$ and q will never be suspected again, a contradiction.

Hence, by (1) and (2) there is a time t'' after which every process that crashes is permanently suspected by p_{leader}, and no correct process is suspected by p_{leader} after t''. Then, by Tasks 1 and 5, and the fairness of the output links of p_{leader}, eventually every correct process will permanently agree with p_{leader} in the set of suspected processes. This gives us the two properties required by $\diamond\mathcal{P}$: eventually every process that crashes is permanently suspected by every correct process (strong completeness), and there is a time after which correct processes are not suspected by any correct process (eventual strong accuracy). □

If we assume that most of the time the $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ provides a unique leader, then the cost of this

```
Every process p executes the following:

suspected_p ← ∅                                          {suspected_p provides the properties of ◇P}
for all q ∈ Π                                            {Δ_p(q) denotes the duration of p's time-out interval for q}
  Δ_p(q) ← default time-out interval

cobegin

|| Task 1: repeat periodically
  if D.trusted_p = p then
    send suspected_p to the rest of processes

|| Task 2: repeat periodically
  if D.trusted_p ≠ p then
    send "p-is-alive" to D.trusted_p

|| Task 3: repeat periodically
  if D.trusted_p = p then
    for all q ∈ Π, q ≠ p:
      if q ∉ suspected_p and
        p did not receive "q-is-alive" during the last Δ_p(q) ticks of p's clock then
        suspected_p ← suspected_p ∪ {q}                  {p times-out on q: it suspects q has crashed}

|| Task 4: when receive "q-is-alive" for some q
  if D.trusted_p = p and q ∈ suspected_p then            {p knows that it prematurely timed-out on q}
    suspected_p ← suspected_p − {q}                      {1. p repents on q, and}
    Δ_p(q) ← Δ_p(q) + 1                                  {2. p increases its time-out period for q}

|| Task 5: when receive suspected_q for some q
  if D.trusted_p = q then
    suspected_p ← suspected_q − {p}                      {p adopts suspected_q}

coend
```

Fig. 2. Transforming the $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ into a $\diamond\mathcal{P}$ failure detector in a model of partial synchrony.

transformation algorithm in terms of the number of messages periodically sent is $2(n-1)$, since the leader process sends a message to the rest of processes, and they send a message to the leader process. Thus, the algorithm has a linear cost. Furthermore, this cost can be reduced in practice. If we assume that the algorithm implementing $\mathcal{D}$ requires the leader process to periodically send a message to the rest of processes (this is the case if we build $\mathcal{D}$ on top of the $\diamond\mathcal{S}$ algorithm proposed in [16], for instance), then the list of suspected processes can be piggy-backed on this message, reducing the number of messages of the transformation algorithm to the half. This may require to revise our assumptions on the synchrony of the links.

Following the previous strategy, we get an extremely efficient implementation of $\diamond\mathcal{P}$ that has a cost of $2(n-1)$ messages periodically sent ($n-1$ of the implementation of the $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ based on [16], and $n-1$ of the transformation algorithm of Fig. 2). This compares favorably to the implementation of $\diamond\mathcal{P}$ proposed by Chandra and Toueg [6], which has a cost of n^2. Also, this is slightly better than the cost ($2n$ messages) of the ring algorithm implementing $\diamond\mathcal{P}$ proposed by Larrea et al. in [15], and this approach has the additional benefit of not suffering of the high latency in crash detection of this algorithm (due to the propagation of the list of suspected processes over the ring).

5. Solving consensus using $\diamond\mathcal{C}$

5.1. The consensus problem

In the Consensus problem, each process initially proposes a value, and all correct processes must reach an irrevocable decision on some common value that is equal to one of the proposed values. Formally, the Consensus problem is defined in terms of two primitives, *propose* and *decide*. When a process executes *propose*(*v*), we say that it *proposes v*. Similarly, when a process executes *decide*(*v*), we say that it *decides v*. The *Consensus* problem must satisfy the following properties.

- *Termination*: Every correct process eventually decides some value.
- *Uniform integrity*: Every process decides at most once.
- *Agreement*: No two correct processes decide differently.
- *Validity*: If a process decides *v*, then *v* was proposed by some process.

Termination defines the liveness property associated with the Consensus problem, while Uniform integrity, Agreement and Validity define its safety properties.

The Agreement property allows faulty processes to decide differently from correct processes. This fact can be sometimes undesirable as it does not prevent an incorrect process to propagate a different decision throughout the system

before crashing. In the *Uniform Consensus* problem, agreement is defined by the following property, which enforces the same decision on any process that decides:

- *Uniform agreement*: No two processes (correct or faulty) decide differently.

It has been shown in [10] that any algorithm that solves Consensus using a failure detector of class $\diamond\mathcal{S}$, also solves Uniform Consensus. Since every failure detector of class $\diamond\mathcal{C}$ includes a failure detector of class $\diamond\mathcal{S}$, we in fact consider the Uniform Consensus problem here.

5.2. The algorithm

In this section, we present an algorithm that solves Uniform Consensus using an eventually consistent failure detector. We assume the system model defined in Section 2. In addition, we assume that the system is augmented with a failure detector $\mathcal{D}$ of class $\diamond\mathcal{C}$, to which processes have access. Finally, we assume that a majority of processes are correct, i.e., do not crash. Thus, if we denote by f the number of processes that can fail, we assume $f < n/2$. This is a necessary requirement to solve Consensus using $\diamond\mathcal{C}$ in asynchronous systems. This can be derived from the relationship between $\diamond\mathcal{P}$ and $\diamond\mathcal{C}$, and Theorem 6.3.1 in [6], which shows that to solve Consensus even with $\diamond\mathcal{P}$ there must be a majority of correct processes. We also assume that the value of f is not known.

Our algorithm is an adaptation of the $\diamond\mathcal{S}$-Consensus algorithm of Chandra and Toueg (where the communication pattern is centralized: all the messages of a round are either sent by the coordinator or received by the coordinator). Figs. 3 and 4 present the algorithm in detail. It is made of a main task (Fig. 3) and 3 additional tasks (Fig. 4). From these 3 additional tasks, the two first tasks are necessary to ensure that a coordinator will never block during the computation, while the third task is used to take the decision.

Each process runs an instance of this algorithm, which proceeds in asynchronous rounds. As the $\diamond\mathcal{S}$-Consensus algorithm of Chandra and Toueg [6], it goes through three asynchronous epochs, each of which may span several rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked*: no other decision value is possible. In the third epoch, processes decide the locked value.

Each round of the main task is divided into five asynchronous phases. In Phase 0, every process determines its coordinator for the round. A process p becomes its own coordinator for the round if it is the process returned by $\mathcal{D}.trusted_p$. A coordinator announces itself by sending a *coordinator* message to the rest of processes. A process becomes a non-coordinator, i.e., a participant, if it receives in Phase 0 a message from a coordinator, which becomes its coordinator for the round.[2] In Phase 1, every process sends its current estimate of the decision value time-stamped with the round number in which it adopted this estimate, to its coordinator. Also after Phase 0 and concurrently with the main algorithm, each process sends a *null* estimate to any other coordinator of the current or previous rounds (first task of Fig. 4).

In Phase 2, each coordinator tries to gather a majority of estimates. If it succeeds, then it selects an estimate with the largest time-stamp and sends it to all the processes as a proposition. On the other hand, if it does not receive a majority of estimates then it sends a *null* proposition to all processes. In Phase 3, each process waits for a proposition from its coordinator. However, it also stops waiting if it suspects its coordinator or if it receives a non-null proposition from some other coordinator. If the process receives a non-null proposition from some coordinator (including its own), then it adopts it and sends an *ack* message to this coordinator. If the process receives a *null* proposition from its coordinator, it stops waiting and passes to the next phase. Finally, if the process suspects its coordinator, it sends a *nack* message to it. After this phase and concurrently with the main algorithm, each process sends a *nack* message to any late coordinator from which it receives a non-null proposition for the current or previous rounds (second task of Fig. 4). Finally, in Phase 4 the coordinator that succeeded in Phase 2 and sent a non-null proposition (if any, and as we will see at most one) tries to gather a majority of *ack* messages. If it succeeds, then it knows that a majority of processes adopted its proposition as their new estimate. Consequently, this coordinator R-broadcasts a request to decide its proposition. At any time, if a process R-delivers such a request, it decides accordingly.

Note that in Phases 2 and 4 the coordinator queries the failure detector and tries to get a majority of "useful" or "positive" replies (i.e., *estimate* messages in Phase 2 and *ack* messages in Phase 4), without blocking thanks to the strong completeness of the sets $D.suspected_p$. More precisely, in Phase 2 instead of waiting for just a majority of replies, each coordinator also waits for a reply from every process it does not suspect. This way additional valid estimates can be received, allowing maybe the coordinator to succeed in the current round. A similar strategy is used in Phase 4 when waiting for $ack/nack$ messages: once a majority of messages are received, the coordinator waits until every process that is not suspected replies. This way, even if *nack* messages are received among the first majority, decision can still be taken if enough additional *ack* messages are received.

If we would like to follow a similar strategy using an Ω failure detector instead of a $\diamond\mathcal{C}$ failure detector, after the reception of a majority of messages the coordinator would never wait for more messages, because in order to satisfy

[2] If a process p waiting in Phase 0 of a round r receives first a *coordinator* message for a round $r' > r$, then p advances to round r'.

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

  while $state_p = undecided$ {*Rotate until decision is reached*}
    $chosen_p \leftarrow false$
    $replied_p \leftarrow false$
    $r_p \leftarrow r_p + 1$

    Phase 0: {*Each process determines its coordinator for the round*}
      wait until [$p = \mathcal{D}.trusted_p$ **or** for a process q: received $(q, r_q, coordinator)$ such that $(r_q \geq r_p)$] {*Query* $\mathcal{D}$}
      if [for a process q: received $(q, r_q, coordinator)$ such that $(r_q \geq r_p)$] **then**
        $c_p \leftarrow q$
        $r_p \leftarrow r_q$
      else
        $c_p \leftarrow p$
        send $(p, r_p, coordinator)$ to the rest of processes
      $chosen_p \leftarrow true$

    Phase 1: {*Each process p sends* $estimate_p$ *to its current coordinator*}
      send $(p, r_p, estimate_p, ts_p)$ to c_p

    Phase 2: {*Each coordinator tries to gather* $\lceil \frac{(n+1)}{2} \rceil$ *estimates to propose a new estimate*}
      if $p = c_p$ **then**
        wait until [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ **or** received $(q, r_p, null\_estimate, 0)$] **and**
            [for every process q: received $(q, r_p, estimate_q, ts_q)$ **or** received $(q, r_p, null\_estimate, 0)$ **or**
              $(q \in \mathcal{D}.suspected_p)$]
        $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
        if [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$] **then**
          $decidable_p \leftarrow true$
          $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
          $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
          send $(p, r_p, estimate_p)$ to all
        else {p *received null_estimate from some process*}
          $decidable_p \leftarrow false$
          send $(p, r_p, null\_estimate)$ to all

    Phase 3: { *Each process waits for a new estimate proposed by a coordinator or to receive null_estimate from its coordinator or to suspect it* }
      wait until [received $(c_p, r_p, null\_estimate)$ from c_p **or** $c_p \in \mathcal{D}.suspected_p$ **or** for a process q: received $(q, r_p, estimate_q)$]
      if [for a process q: received $(q, r_p, estimate_q)$] **then** {p *received* $estimate_q$ *from a process* q}
        $estimate_p \leftarrow estimate_q$
        $ts_p \leftarrow r_p$
        send (p, r_p, ack) to q
      else if [received $(c_p, r_p, null\_estimate)$ from c_p] **then** {p *received null_estimate from* c_p}
        discard message
      else {p *suspects that* c_p *crashed*}
        send $(p, r_p, nack)$ to c_p
      $replied_p \leftarrow true$

    Phase 4: { *The coordinator that can still decide (if any) waits for* $\lceil \frac{(n+1)}{2} \rceil$ *replies. If they indicate that* $\lceil \frac{(n+1)}{2} \rceil$ *processes adopted its estimate, the coordinator R-broadcasts a decide message* }
      if $(p = c_p)$ **and** $(decidable_p)$ **then**
        wait until [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received (q, r_p, ack) **or** received $(q, r_p, nack)$] **and**
            [for every process q: received (q, r_p, ack) **or** received $(q, r_p, nack)$ **or** $(q \in \mathcal{D}.suspected_p)$]
        if [for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received (q, r_p, ack)] **then**
          R-broadcast $(p, r_p, estimate_p, decide)$

Fig. 3. Solving Consensus using any $\mathcal{D} \in \diamond\mathcal{C}$.

when received $(q, r_q, coordinator)$ from q such that $((r_q < r_p)$ **or** $((r_q = r_p)$ **and** $(chosen_p)))$
send $(p, r_q, null\_estimate, 0)$ to q

when received $(q, r_q, estimate_q)$ from q such that $((r_q < r_p)$ **or** $((r_q = r_p)$ **and** $(replied_p)))$
send $(p, r_q, nack)$ to q

when $R\text{-}deliver\ (q, r_q, estimate_q, decide)$ {*If p R-delivers a decide message, p decides accordingly*}
if $state_p = undecided$ **then**
$decide\ (estimate_q)$
$state_p \leftarrow decided$

Fig. 4. Separate tasks for replying to late coordinators and taking the decision.

strong completeness with the Ω detector all processes but the leader must be suspected.

5.3. Correctness proof of the consensus algorithm

In this section, we prove the correctness of the algorithm presented. Note that some parts of the proof rely on the properties satisfied by a communication primitive called *Reliable Broadcast*. We refer the reader to [6] for details about this primitive.

Lemma 1. *In any round r, at most one coordinator c will send a non-null estimate, i.e., a message of type* (*c*, *r*, $estimate_c$), *to all processes at the end of Phase* 2.

Proof. From the algorithm, it is clear that each process sends its estimate to only one coordinator in Phase 1. A coordinator must gather a majority of such estimates in order to send a non-null estimate at the end of Phase 2. Thus, at most one coordinator can gather such a majority and send a non-null estimate. □

From the above lemma and the algorithm, it is easy to see that in any round r at most one coordinator c will R-broadcast a (*c*, *r*, $estimate_c$, *decide*) message in Phase 4. [3]

Lemma 2. *No two processes decide differently.*

Proof. If no process ever decides, the lemma is trivially true. If any process decides, it must have previously R-delivered a message of type (–, –, –, *decide*). By the uniform integrity property of Reliable Broadcast and the algorithm, a coordinator previously R-broadcast this message. This coordinator must have received at least $\lceil (n+1)/2 \rceil$ messages of type (–, –, *ack*) in Phase 4. Let r be the smallest round number in which at least $\lceil (n+1)/2 \rceil$ messages of type (–, –, *ack*) are sent to a coordinator c in Phase 3. Let $estimate_c$ denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geqslant r$, if a coordinator c' sends $estimate_{c'}$ in Phase 2 of round r', then $estimate_{c'} = estimate_c$.

The proof is by induction on the round number. From Lemma 1 and the fact that c sends $estimate_c$ at the end of Phase 2 of round r, any other coordinator of round r sends *null_estimate* at the end of Phase 2 of round r. Thus, the claim holds for $r' = r$. Now assume that the claim holds for all r', $r \leqslant r' < k$. We will show that the claim holds for $r' = k$, that is, if c_k is a coordinator of round k that sends $estimate_{c_k}$ in Phase 2, then $estimate_{c_k} = estimate_c$.

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

In Phase 2 of round k, c_k executes $estimate_{c_k} \leftarrow estimate_q$ where (*q*, *k*, $estimate_q$, *t*) is in $msgs_{c_k}[k]$ (remember that we assume that c_k sends $estimate_{c_k}$ in Phase 2 of round k). From Fig. 3, it is clear that q adopted $estimate_q$ as its estimate in Phase 3 of round t. Thus, a coordinator of round t sent $estimate_q$ to q in Phase 2 of round t. Since $r \leqslant t < k$, by the induction hypothesis, $estimate_q = estimate_c$. Thus, c_k sets $estimate_{c_k} \leftarrow estimate_c$ in Phase 2 of round k. This concludes the proof of the claim.

We now show that if a process decides a value, then it decides $estimate_c$. Suppose that some process p R-delivers (*q*, r_q, $estimate_q$, *decide*), and thus decides $estimate_q$. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast (*q*, r_q, $estimate_q$, *decide*) in Phase 4 of round r_q. From Fig. 3, q must have received $\lceil (n+1)/2 \rceil$ messages of type (–, r_q, *ack*) in Phase 4 of round r_q. By the definition of r, $r \leqslant r_q$. From the above claim, $estimate_q = estimate_c$. □

[3] This is achieved through the use of the $decidable_p$ boolean variable.

Lemma 3. *Every correct process eventually decides some value.*

Proof. There are two possible cases:

(1) Some correct process decides. It must have R-delivered some message of type (–, –, –, *decide*). By the agreement property of Reliable Broadcast, all correct processes eventually R-deliver this message and decide.
(2) No correct process decides. We claim that no correct process remains blocked forever at one of the **wait** statements. The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the **wait** statements. Therefore, every correct process reaches Phase 0 of round r. From the eventual leader election property of $\mathcal{D}$, eventually some correct process c will become coordinator (i.e., $c = \mathcal{D}.trusted_p$) in some round $r' \geqslant r$. Hence, c will send a message of type (c, r', *coordinator*) to the rest of processes. Thus, all correct processes reach the end of Phase 0 of round r, either by becoming coordinator, or by receiving a message of type (–, r', *coordinator*) from a coordinator of round $r' \geqslant r$. If $r' > r$ this directly translates the process to round r'. Thus, no correct process blocks forever at the **wait** statement of Phase 0.

It is also clear from the algorithm that all correct processes reach the end of Phase 1 of round r: they all send a message of type (–, r, *estimate*, –) to their current coordinator. Let us consider now Phases 2 and 3. For each coordinator c of round r, there are two cases to consider:

(a) From the first task of Fig. 4 and since a majority of the processes are correct, eventually c receives at least $\lceil (n+1)/2 \rceil$ messages, either of type (–, r, *estimate*, –) or (–, r, *null_estimate*, 0). Also, since the sets $\mathcal{D}.suspected_p$ satisfy strong completeness, for every process q, c will eventually receive a message from q or will suspect q. Finally, according to the messages received c replies by sending (c, r, $estimate_c$) or (c, r, *null_estimate*). Thus, c does not block forever at the **wait** statement in Phase 2.
(b) c crashes.

In the first case, every correct process p eventually receives (c, r, $estimate_c$) from a coordinator c (which can be its coordinator) or (c_p, r, *null_estimate*) from its coordinator c_p. In the second case, since the sets $\mathcal{D}.suspected_p$ satisfy strong completeness, for every correct process p there is a time after which its coordinator c_p is permanently suspected by p, that is, $c_p \in \mathcal{D}.suspected_p$. Thus in either case, no correct process blocks at the **wait** statement in Phase 3. From Phase 3 of the algorithm and the second task of Fig. 4, it must be clear that every correct process that receives a non-null estimate from a coordinator c, replies to it with a message of type (–, r, *ack*) or (–, r, *nack*). Since there are at least $\lceil (n+1)/2 \rceil$ correct processes, a coordinator that sent a non-null estimate at the end of Phase 2 will eventually receive at least $\lceil (n+1)/2 \rceil$ such messages. Also, since the sets $\mathcal{D}.suspected_p$ satisfy strong completeness, for every process q, that coordinator will eventually receive a message from q or will suspect q. Note that only the coordinators that sent a non-null estimate at the end of Phase 2 execute the **wait** statement of Phase 4. Thus, no coordinator can block at the **wait** statement of Phase 4. This shows that all correct processes complete round r—a contradiction that completes the proof of our claim.

Since the values $\mathcal{D}.trusted_p$ satisfy the eventual leader election property, there is a correct process q and a time t such that every correct process permanently *trusts* q after t, i.e. for every correct process p: $\mathcal{D}.trusted_p = q$ after t. Let $t' > t$ be a time such that all faulty processes have crashed. From the above claim, any round that started before t' will eventually end. Let r be a round that starts after that happens. Clearly, q is the only possible coordinator of round r. In Phase 0 of round r, q sends (q, r, *coordinator*) to all correct processes except itself. Thus, in Phase 0 every correct process except q receives (q, r, *coordinator*) from q and sets q as its coordinator and r as its current round. In Phase 1, all correct processes send their estimates to q. In Phase 2, q receives $\lceil (n+1)/2 \rceil$ such estimates, and sends (q, r, $estimate_q$) to all processes. In Phase 3, since q is not suspected by any correct process after time t', every correct process waits for q's estimate, eventually receives it, and replies with an *ack* to q. Thus, in Phase 4, q receives $\lceil (n+1)/2 \rceil$ messages of type (–, r, *ack*) (and no message of type (–, r, *nack*)), and R-broadcasts (q, r, $estimate_q$, *decide*). By the validity and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message and *decide*—a contradiction. Thus, case (2) is impossible, and this concludes the proof of the lemma. □

Theorem 2. *The algorithm of Figs. 3 and 4 solves Uniform Consensus using a $\diamond\mathcal{C}$ failure detector $\mathcal{D}$ in asynchronous systems with $f < n/2$.*

Proof. Lemmas 2 and 3 show that the algorithm of Figs. 3 and 4 satisfies the uniform agreement and termination properties of Consensus, respectively. From the algorithm, it is clear that no process decides more than once, and hence the uniform integrity property holds. From the algorithm it is also clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be a value proposed by some process. Thus, uniform validity of Consensus is also satisfied. □

5.4. Performance analysis

In this section, we analyze the performance of the proposed $\diamond\mathcal{C}$-based Consensus protocol, and compare it with

the $\diamond\mathcal{S}$-based protocol proposed by Chandra and Toueg [6], and the Ω-based protocol proposed by Mostefaoui and Raynal [20]. First, we compare them in terms of the number of communication steps per round, and the number of messages exchanged per round, showing that there is an inherent trade-off between these two measures. We show then that our $\diamond\mathcal{C}$-Consensus protocol performs better in the number of rounds required to solve Consensus than the $\diamond\mathcal{S}$-based protocols, due to the fact that the later rely on the rotating coordinator paradigm while ours do not. The protocol of Mostefaoui and Raynal exhibits also this advantage.

If we look to the number of communication steps (phases) per round, the protocol proposed in this paper has five phases per round. The protocol of Chandra and Toueg has four phases per round, and the protocol of Mostefaoui and Raynal has three phases per round.

Concerning the number of messages exchanged in each round, and considering the "normal" case in which there are no crashes and the failure detector does not make any mistake, our $\diamond\mathcal{C}$-Consensus protocol requires $4n$, i.e., $\Theta(n)$, messages per round. Similarly, the protocol of Chandra and Toueg requires $3n$, i.e., $\Theta(n)$, messages per round, and the protocol of Mostefaoui and Raynal requires $3n^2$, i.e., $\Theta(n^2)$, messages per round. (In this protocol, each one of its three phases begins with a message broadcast.) In all cases, we have not considered the messages involved in the *Reliable Broadcast* primitive used to communicate the decision to all processes. Note also that Phase 0 of our $\diamond\mathcal{C}$-Consensus protocol could require $\Omega(n^2)$ messages in the "bad" case in which all the processes consider themselves as the leader.

Clearly, there exists a trade-off between the number of messages and the number of communication steps per round of the protocols. For example, we could reduce the number of phases of our $\diamond\mathcal{C}$-Consensus protocol by merging Phases 0 and 1 in the following way: each process sends its estimate to its leader (obtained by querying the failure detector), and it also sends *null_estimate* to every other process. This reduction on the number of phases has the cost of augmenting the number of messages, which becomes $\Omega(n^2)$ instead of $\Theta(n)$.

Finally, concerning the number of rounds required to solve Consensus, note that our $\diamond\mathcal{C}$-based Consensus protocol and the Ω-based protocol of Mostefaoui and Raynal do not use the rotating coordinator paradigm. Instead, in both cases the eventual leader election functionality provided by the failure detector is exploited. As a result, in the case of stability of the failure detector (i.e., it returns the same trusted process to all processes), Consensus is solved in only one round, providing early consensus.

As shown in the following theorem, in any $\diamond\mathcal{S}$-Consensus algorithm based on the rotating coordinator paradigm (like, for instance, the Chandra and Toueg protocol), the number of rounds can be $\Omega(n)$ once the failure detector is stable (i.e., there is one correct process that is never suspected by any process), until a correct and not-suspected process becomes coordinator of a round. We assume here that Consensus cannot be reached if all processes suspect of the potential coordinator of a round.

Theorem 3. *For any algorithm that solves Consensus with a $\diamond\mathcal{S}$ detector based on the rotating coordinator paradigm there is a run that requires n rounds after the failure detector is stable.*

Proof. We will consider here a run in which the rounds run synchronously, i.e., all processors start and end the same round simultaneously. The $\diamond\mathcal{S}$ failure detector we use satisfies that, before some time t all processes suspect each other, and at t a given correct process p stops being suspected by every process, reaching the stability of the failure detector. We get to choose t and p.

Note first that in any algorithm all processes must be potential coordinators over and over again until Consensus is reached. Otherwise, we can choose a processor p that does not satisfy this and some time t after p is potential coordinator for the last time, and Consensus is never reached.

Then, there is at least one process for which consecutive rounds as potential coordinator, r_0 and r_1, are at least n rounds apart. Let us make p any such process, and t the time round r_0 ends. Then, the claim follows, since until round $r_1 \geqslant r_0 + n$ Consensus cannot be reached. □

However, even if the detector is not stable, Consensus can be reached if the appropriate conditions are met. As we mentioned above, to improve the chances of reaching Consensus, we have introduced in this protocol an interesting feature. The two wait statements in Phases 2 and 4 block the execution of the coordinator until it receives a majority of replies and it has received a reply from each process it does not suspect. Then, in both cases it is enough to have a majority of positive replies to continue toward a decision, while many of the other replies could be negative. This feature may mean the difference between deciding or not in a given round, compared, for instance, with the Chandra and Toueg protocol, in which a coordinator only waits for a majority of replies (the first $\lceil (n+1)/2 \rceil$ replies) and does not decide in the round if any reply is not positive. This feature also shows how the possible higher accuracy of a detector in $\diamond\mathcal{C}$ could be useful versus a detector in Ω. In the protocol of Mostefaoui and Raynal all they could do was to wait for $n - f$ replies, since the detector only provides information about one process. If there is not a good knowledge about f, their protocol could not decide in a round with a majority of positive replies. For instance, if all it is known is that $f < n/2$, a single negative reply in the first $\lceil (n+1)/2 \rceil$ replies prevents from deciding in that round.

6. Conclusions

In this paper, we have proposed a novel class of unreliable failure detectors, called Eventually Consistent and denoted $\diamond\mathcal{C}$. We have studied the relationship between $\diamond\mathcal{C}$ and other failure detector classes. We have also proposed an efficient algorithm transforming $\diamond\mathcal{C}$ into $\diamond\mathcal{P}$ in models of partial synchrony.

The failure detectors of class $\diamond\mathcal{C}$ combine a failure detection capability with an eventual leader election functionality. These properties can be very useful. To demonstrate it, we have presented an efficient algorithm for solving Consensus based on an eventually consistent failure detector. The class $\diamond\mathcal{C}$ allows the algorithm to use a more selective approach to choose a coordinator. This approach allows our algorithm to reach consensus in one single round in stability, while $\diamond\mathcal{S}$-Consensus algorithms based on the rotating coordinator paradigm may require $\Omega(n)$ rounds.

Acknowledgements

We would like to thank the anonymous reviewers, and specially Reviewer III, who did a great work in the review of this paper and whose comments and suggestions have dramatically improved the quality of the paper. We are also grateful to André Schiper and Michel Raynal for helpful comments.

References

[1] M. Aguilera, W. Chen, S. Toueg, Heartbeat: a timeout-free failure detector for quiescent reliable communication, in: Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97), Saarbrucken, Germany, Lecture Notes in Computer Science, vol. 1320, Springer, Berlin, September 1997, pp. 126–140.

[2] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, Stable leader election, in: Proceedings of the 15th International Symposium on Distributed Computing (DISC'2001), Lisbon, Portugal, Lecture Notes in Computer Science, vol. 2180, Springer, Berlin, October 2001, pp. 108–122.

[3] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, S. Toueg, On implementing Ω with weak reliability and synchrony assumptions, in: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'2003), Boston, Massachusetts, July 2003, pp. 306–314.

[4] M. Aguilera, S. Toueg, B. Deianov, Revisiting the weakest failure detector for uniform reliable broadcast, in: Proceedings of the 13th International Symposium on Distributed Computing (DISC'99), Bratislava, Slovak Republic, Lecture Notes in Computer Science, vol. 1693, Springer, Berlin, September 1999, pp. 19–33.

[5] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (4) (1996) 685–722.

[6] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.

[7] F. Chu, Reducing Ω to $\diamond\mathcal{W}$, Inform. Process. Lett. 67 (1998) 289–293.

[8] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.

[9] M. Fischer, N. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.

[10] R. Guerraoui, Revisiting the relationship between non-blocking atomic commitment and consensus, in: Proceedings of the Ninth International Workshop on Distributed Algorithms (WDAG'95), Le Mont-Saint-Michel, France, Lecture Notes in Computer Science, vol. 972, Springer, Berlin, September 1995, pp. 87–100.

[11] R. Guerraoui, A. Schiper, Γ-accurate failure detectors, in: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96), Bologna, Italy, Lecture Notes in Computer Science, vol. 1151, Springer, Berlin, October 1996, pp. 269–286.

[12] M. Hurfin, M. Raynal, A simple and fast asynchronous consensus protocol based on a weak failure detector, Distrib. Comput. 12 (4) (1999) 209–223.

[13] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.

[14] M. Larrea, Efficient algorithms to implement failure detectors and solve consensus in distributed systems, Ph.D. Thesis, University of the Basque Country, San Sebastián, October 2000.

[15] M. Larrea, S. Arévalo, A. Fernández, Efficient algorithms to implement unreliable failure detectors in partially synchronous systems, in: Proceedings of the 13th International Symposium on Distributed Computing (DISC'99), Bratislava, Slovak Republic, Lecture Notes in Computer Science, vol. 1693, Springer, Berlin, September 1999, pp. 34–48.

[16] M. Larrea, A. Fernández, S. Arévalo, Optimal implementation of the weakest failure detector for solving consensus, in: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000), Nurenberg, Germany, October 2000, pp. 52–59.

[17] M. Larrea, A. Fernández, S. Arévalo, Eventually consistent failure detectors, in: Proceedings of the 10th Euromicro Workshop on Parallel Distributed, and Network-based Processing (PDP'2002), Las Palmas de Gran Canaria, Spain, January 2002Also in Brief Announcements of the 14th International Symposium on Distributed Computing (DISC 2000), Toledo, Spain, October 2000, pp. 91–98.

[18] A. Mostefaoui, M. Raynal, Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach, in: Proceedings of the 13th International Symposium on Distributed Computing (DISC'99), Bratislava, Slovak Republic, Lecture Notes in Computer Science, vol. 1693, Springer, Berlin, September 1999, pp. 49–63.

[19] A. Mostefaoui, M. Raynal, Unreliable failure detectors with limited scope accuracy and an application to consensus, in: Proceedings of the 19th International Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS'99, Lecture Notes in Computer Science, Springer, Berlin, December 1999, pp. 329–340.

[20] A. Mostefaoui, M. Raynal, Leader-based consensus, Parallel Process. Lett. 11 (1) (2001) 95–107 Also IRISA Technical Report 1372, December 2000.

[21] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.

[22] A. Schiper, Early consensus in an asynchronous system with a weak failure detector, Distrib. Comput. 10 (3) (1997) 149–157.

Mikel Larrea is an assistant professor at the Universidad del País Vasco in San Sebastián, Spain, since 2001. Previously, he was on the faculty of the Universidad Pública de Navarra. He graduated in computer science from the Swiss Federal Institute of Technology in Lausanne, Switzerland, in 1995. He got a Ph.D. in computer science from the Universidad del País Vasco in 2000. His research interests include distributed systems and algorithms, fault tolerance, and group communication systems.

Antonio Fernández is an associate professor at the Universidad Rey Juan Carlos in Madrid, Spain, since 1998. Previously, he was on the faculty of the Universidad Politécnica de Madrid. He graduated in Computer Science from the Universidad Politécnica de Madrid in 1991. He got a Ph.D. in Computer Science from the University of Southwestern Louisiana in 1994 and was a postdoc at the Massachusetts Institute of Technology from 1995 to 1997. His research interests include data communications, computer networks, parallel and distributed processing, algorithms, and discrete and applied mathematics.

Sergio Arévalo received the MS and Ph.D. in computer science in Spain from the Technical University of Madrid, in 1983 and 1988 respectively. Presently, he is a professor in the Department of Computer Science, Universidad Rey Juan Carlos, Madrid, Spain. He was invited researcher at AT&T Bell Laboratories, Murray Hill, NJ, in 1987 and 1988. He has been also post-doctoral research fellow in 1989 at the European Space Agency, Nordwijk, The Netherlands. His research interests include distributed languages and systems, fault-tolerant computing, and operating systems. He is a member of the ACM since 1983.