Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems*

Mikel Larrea[1], Sergio Arevalo[2], and Antonio Fernndez[2]

[1] Universidad Pblica de Navarra, 31006 Pamplona, Spain
`mikel.larrea@unavarra.es`
[2] Universidad Rey Juan Carlos, 28933 Mstoles, Spain
`{s.arevalo,a.fernandez}@escet.urjc.es`

Abstract. Unreliable failure detectors, proposed by Chandra and Toueg [2], are mechanisms that provide information about process failures. In [2], eight classes of failure detectors were defined, depending on how accurate this information is, and an algorithm implementing a failure detector of one of these classes in a partially synchronous system was presented. This algorithm is based on all-to-all communication, and periodically exchanges a number of messages that is quadratic on the number of processes. To our knowledge, no other algorithm implementing these classes of unreliable failure detectors has been proposed.
In this paper, we present a family of distributed algorithms that implement four classes of unreliable failure detectors in partially synchronous systems. Our algorithms are based on a logical ring arrangement of the processes, which defines the monitoring and failure information propagation pattern. The resulting algorithms periodically exchange at most a linear number of messages.

1 Introduction

The concept of *unreliable failure detector* was introduced by Chandra and Toueg in [2]. These authors showed how unreliable failure detectors can be used to solve the *Consensus* problem [10] in asynchronous systems. (This was shown to be impossible in a pure asynchronous system by Fischer et al. [7].) Since then, a considerable amount of work has been devoted to study properties of the failure detection abstraction [1,6,9].

From the results of Fischer et al. and those of Chandra and Toueg, it can be derived the impossibility of, in asynchronous systems, implementing failure detectors precise enough to solve Consensus. Chandra and Toueg presented an algorithm that implements an unreliable failure detector in a partially synchronous system. To our knowledge, this is the only proposed algorithm implementing any

* Partially supported by the Basque Government under grant number BFI96.043, and the Madrid Regional Research Council (CAM), contract number $CAM-07T/00112/1998$.

of the classes of unreliable failure detectors defined in [2]. In this paper we present more efficient alternatives to that first algorithm.

1.1 Partial Synchrony

Distributed algorithms can be designed under different assumptions of system behaviors, i.e., system models. One of the main assumptions in which system models can differ is related to the timing aspects. Most models focus on two timing attributes: the time taken for message delivery across a communication channel, and the time taken by a processor to execute a piece of code. Depending on whether these attributes are bounded or not, and on the knowledge of these bounds, they can be classified as synchronous, asynchronous, or partially synchronous [5]. A timing attribute is *synchronous* if there is a known fixed upper bound on it. On the other hand, it is *asynchronous* if there is no bound on it. Finally, a timing attribute is *partially synchronous* if it is neither synchronous nor asynchronous. Dwork et al. [5] consider two kinds of partial synchrony. In the first one, the timing attribute is bounded, but the bound is unknown. In the second one, the timing attribute is bounded and the bound is known, but it holds only after an unknown stabilization interval. Chandra and Toueg [2] propose another kind of partial synchrony, in which the timing attribute is bounded, but the bound is unknown and holds only after an unknown stabilization interval. This will be the model of partial synchrony used in this paper.

Although the asynchronous model (in which at least one of the timing attributes is asynchronous) is attractive for designing distributed algorithms, it is well known that a number of synchronization distributed problems cannot be solved deterministically in asynchronous systems in which processes can fail. For instance, as we mentioned above, Consensus cannot be solved deterministically in an asynchronous system that is subject to even a single process failure [7], while it can be solved in both synchronous and partially synchronous systems [2,4,5]. In fact, the ability to solve these synchronization distributed problems closely depends on the ability to detect failures. In a synchronous system, reliable failure detection is possible. One can reliably detect failures using timeouts. (The timeouts can be derived from the known upper bounds on message delivery time and processing time.) In an asynchronous system, it is impossible to distinguish a failed process from a very slow one. Thus, reliable failure detection is impossible.

However, even if it is sufficient, reliable failure detection is not necessary to solve most of these problems. As we already mentioned, Chandra and Toueg [2] introduced unreliable failure detectors (failure detectors that can make mistakes), and showed how they can be used to solve Consensus and Atomic Broadcast. Guerraoui et al. [8] showed how unreliable failure detectors can be used to solve the Non-Blocking Atomic Commitment problem.

1.2 Unreliable Failure Detectors

An *unreliable failure detector* is a mechanism that provides (possibly incorrect) information about process failures. When it is queried, the failure detector re-

	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.

turns a list of processes believed to have crashed (suspected processes). In [2], 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, which combined yield eight classes of failure detectors.

In this paper we will focus on the two kinds of completeness and two of the four kinds of accuracy defined in [2], which are the following:

– *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.

Note that completeness by itself is not very useful. We can trivially satisfy strong completeness by forcing every process to permanently suspect every other process in the system.

– *Eventual strong accuracy.* Eventually, no correct process is ever suspected by any correct process.
– *Eventual weak accuracy.* Eventually, 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. Out of these, Chandra et al. [3] showed that $\Diamond\mathcal{W}$ is the weakest class of failure detectors required for solving Consensus.

Chandra and Toueg [2] proposed a timeout-based implementation of a $\Diamond\mathcal{P}$ failure detector in a system with partial synchrony (they recognize that, in practice, some synchrony is required to implement the failure detectors they propose). In their algorithm, all processes periodically send a message to every other process in order to inform them that it has not crashed. If there are n processes in the system and $\mathcal{C}$ of them do not crash, at least $n\mathcal{C}$ messages are periodically exchanged with this algorithm. We do not know of any other implementation of these classes of failure detectors.

1.3 Our Results

In this paper, we present a family of algorithms that implement unreliable failure detectors of the four classes defined in the previous section, in partially synchronous systems. Our algorithms have been designed, and are presented, in a gradual way. First, we present an algorithm that provides weak completeness. Next, we show how to extend this algorithm to provide eventual weak accuracy. This extended algorithm implements a $\diamond\mathcal{W}$ failure detector. Next, we present two other extensions which strengthen the accuracy and the completeness, respectively, implementing the stronger failure detectors.

In all these algorithms, each correct process monitors only one other process in a cyclic fashion. The monitoring process performs this task by repeatedly polling the monitored process. Each polling involves only two messages exchanged between the monitoring and monitored processes. If the pollings were done periodically, a total of no more than $2n$ messages would be periodically exchanged. Eventually, this amount becomes at most $2\mathcal{C}$, which is a significant improvement over the at least $n\mathcal{C}$ messages of the previous algorithm (Chandra and Toueg's).

The rest of the paper is organized as follows. The next section describes our model of distributed system. In Section 3, we present a basic algorithm that provides weak completeness. In Section 4, we present an extension to the basic algorithm that provides eventual weak accuracy. In Section 5, we present another extension that provides eventual strong accuracy. In Section 6, we present an extension to the previous algorithms that provides strong completeness, while preserving accuracy. In Section 7, we study the performance of our algorithms in terms of the number and the size of the messages periodically exchanged. Finally, Section 8 summarizes the conclusions and presents future lines of work.

2 The Model

2.1 System Model

Our model of distributed system consists of a set Π of n processes, $\Pi = \{p_1, \ldots, p_n\}$, that communicate by exchanging messages. Every pair of processes is assumed to be connected by a reliable communication channel.

Processes can fail by *crashing*, that is, by prematurely halting. Crashed processes do not recover. In every execution of the system we identify two complementary subsets of Π: the subset of processes that do not fail, denoted *correct*, and the subset of processes that do fail, denoted *crashed*. We use $\mathcal{C}$ to denote the number of correct processes in the system, which we assume is at least one, i.e., $\mathcal{C} = |correct| > 0$. For every process p in *crashed* we use $Tcrash_p$ to denote the instant at which p crashes.

In the algorithms presented in this paper we consider the processes $p_1, \ldots, p_n$ arranged in a logical ring. This arrangement is known by all the processes. Without loss of generality, process p_i is followed in the ring by process $p_{(i \bmod n)+1}$. In general, we use $succ(p)$ to denote the process that follows process p in the ring, and $pred(p)$ to denote the process that precedes process p in the ring. Finally, we use $corr\_succ(p)$ and $corr\_pred(p)$ to denote the closest correct (i.e., belonging to the subset *correct*) successor and predecessor of p in the ring, respectively.

We consider the model of *partial synchrony* proposed by Chandra and Toueg [2]. In this model, there are bounds on both message delivery time and processing time, but these bounds are not known and only hold after an unknown, but finite, stabilization interval. We shall use T_s to denote the ending instant of this stabilization interval in the execution of interest. We also denote by Δ_{msg} the maximum time interval, after stabilization, since a process sends a message and that message is delivered and processed by its destination process (assuming that both the sender and the destination have not failed). Clearly, Δ_{msg} depends on the existing bounds on both message delivery time and processing time. Note that the exact value of Δ_{msg} exists, but it is unknown.

2.2 Implementation of Failure Detectors

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. Each module maintains a list of the processes it suspects to have crashed. These lists can differ from one module to another at a given time. We denote by L_p (G_p in Section 6) the list of suspected processes of the failure detection module attached to process p. Clearly, the contents of the list L_p (G_p) can be different at different times. We use $L_p(t)$ ($G_p(t)$ in Section 6) to denote the contents of L_p (G_p) at time t. A process p interacts only with its local failure detection module in order to get the current list of suspected processes.

In this paper, we only describe the behavior of the failure detection modules in order to implement a failure detector, but not the behavior of the processes they are attached to. For this reason, in the rest of the paper we will mostly use the term *process* instead of *failure detection module*. We consider that a process cannot crash independently of its attached failure detection module.

In any algorithm that implements any of the failure detector classes defined in Section 1.2, it is required that some processes monitor other processes. Monitoring allows a process to detect whether another process has crashed and to take proper action if so (usually suspect it). Clearly, there are several possible ways to implement the monitoring. Examples are the monitored process sending an I-AM-ALIVE message (a *heartbeat*) to the monitoring process or the later *polling* the former for such a message. In any case, the only way a process can show it has not crashed is by sending messages to those monitoring it. Hence, any monitoring protocol requires that the monitored process sends messages to the monitoring process.

Our algorithms use pollings instead of only sending heartbeats, because the former allow a finer control of the monitoring. To monitor process q, a process p sends an ARE-YOU-ALIVE? message to q and waits for an I-AM-ALIVE message from it. As soon as q receives the ARE-YOU-ALIVE? message, it sends the I-AM-ALIVE message to p. We will denote by $\Delta_{rtt} = 2\Delta_{msg}$ the maximum monitoring round-trip time after stabilization, i.e., the maximum time, after T_s, elapsed between the sending of an ARE-YOU-ALIVE? message to a correct process, and the reception and processing of the corresponding I-AM-ALIVE reply message.

Since a monitoring process p does not know Δ_{rtt}, it has to use an estimated value (timeout) that tells how much time it has to wait for the reply from the monitored process q. This time value is denoted by $\Delta_{p,q}$. Then, if after $\Delta_{p,q}$ time p did not receive the reply from q, it suspects that q has crashed. We need to allow these time values to vary over time in our algorithms. We use $\Delta_{p,q}(t)$ to denote the value of $\Delta_{p,q}$ at time t.

3 A Basic Algorithm that Provides Weak Completeness

In this section, we present an algorithm that will be used as a framework for all the failure detector implementations presented in this paper. This first algorithm satisfies the weak completeness property. In the following sections we will extend the algorithm to satisfy also eventual weak accuracy, eventual strong accuracy, and strong completeness. This algorithm is presented here for the sake of clarity but is not very useful by itself, since it does not satisfy any of the accuracy properties previously defined.

The algorithm executes as follows: initially, every process starts monitoring its successor in the ring. If a process p does not receive the reply from the process q it is monitoring, then p suspects that q has crashed, and starts monitoring the successor of q in the ring. This monitoring scheme is repeated, so that p always suspects all processes in the ring between itself and the process it is monitoring (not included). If, later on, p receives a message from a suspected process q while it is monitoring another process r, then p stops suspecting q and all the processes between q and r in the ring, and starts monitoring q again.

Fig. 2 presents the algorithm in detail. Each process p has a variable $target_p$ which holds the process being monitored by p at a given time. As we said above, all processes between p and $target_p$ in the ring (and only them) are suspected by p, and these are the only processes included in the list L_p of suspected processes of p. (Initially, no process is suspected, i.e., $\forall p : L_p(0) = \emptyset$.) The $mutex_p$ variable is used to avoid race conditions in process p.

We now show that weak completeness holds with this algorithm. Given an incorrect process p, the following theorem states that it will be permanently suspected by $corr\_pred(p)$ (the first correct process preceding p in the ring).

Theorem 1. *$\exists t_0 : \forall p \in crashed$, p has failed at time t_0 and $\forall t \geq t_0, p \in L_{corr\_pred(p)}(t)$.*

Proof. Let p be a process that crashes. We claim that p will be permanently included in $L_{corr\_pred(p)}$. The proof uses strong induction on the distance from $corr\_pred(p)$ to p. Let first consider that such distance is 1, i.e., $corr\_pred(p) = pred(p)$. Before p fails, $corr\_pred(p)$ and p exchange ARE-YOU-ALIVE? and I-AM-ALIVE messages (see Fig. 2). Eventually p crashes, and there is an ARE-YOU-ALIVE? message sent by $corr\_pred(p)$ that reaches p after $Tcrash_p$. Since p has

Every process p executes:

```
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

**cobegin**
|| Task 1:
  **loop**
    wait($mutex_p$)
    send ARE-YOU-ALIVE? to $target_p$
    $t_{out} \leftarrow \Delta_{p,target_p}$
    $received \leftarrow false$
    signal($mutex_p$)
    delay $t_{out}$
    wait($mutex_p$)
    **if not** $received$
      $L_p \leftarrow L_p \cup \{target_p\}$
      $target_p \leftarrow succ(target_p)$
    **end if**
    signal($mutex_p$)
  **end loop**

|| Task 2:
  **loop**
    receive message $m$ from a process $q$
    wait($mutex_p$)
    **case**
      $m =$ ARE-YOU-ALIVE?:
        send I-AM-ALIVE to $q$
        **if** $q \in L_p$
          $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
          $target_p \leftarrow q$
          $received \leftarrow true$
        **end if**
      $m =$ I-AM-ALIVE:
        **case**
          $q = target_p$:
            $received \leftarrow true$
          $q \in L_p$:
            $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
            $target_p \leftarrow q$
            $received \leftarrow true$
          **else** discard $m$
        **end case**
    **end case**
    signal($mutex_p$)
  **end loop**
**coend**
```

Fig. 2. Algorithm that provides weak completeness.

already crashed by then, it will never reply to that message. If such a message was sent at time t', then $\Delta_{corr\_pred(p),p}(t')$ time later, $corr\_pred(p)$ will include p in $L_{corr\_pred(p)}$. Since no message will ever be received by $corr\_pred(p)$ from p after that, it will never be removed from $L_{corr\_pred(p)}$.

We will now prove that if the claim holds for any distance $1 \leq d \leq i-1$, it also holds for distance i. Let us assume the distance from $corr\_pred(p)$ to p be $i > 1$. Then, for any process $q \in \{succ(corr\_pred(p)), \ldots, pred(p)\}$, it can be easily seen that $corr\_pred(q) = corr\_pred(p)$ and the distance d from $corr\_pred(p)$ to q verifies $1 \leq d \leq i-1$. Hence, from the induction hypothesis, all processes in $\{succ(corr\_pred(p)), \ldots, pred(p)\}$ will eventually be permanently in $L_{corr\_pred(p)}$. After that, they will never be monitored again by $corr\_pred(p)$. The situation then is similar to the distance-1 case considered above and, by a similar argument, p will eventually be permanently included in $L_{corr\_pred(p)}$.

Corollary 1. *The algorithm of Fig. 2 provides weak completeness.*

4 Extending the Basic Algorithm to Provide Eventual Weak Accuracy

The algorithm presented in the previous section does not satisfy any of the accuracy properties defined in Section 1.2. It does not prevent the erroneous suspicion of any correct process, and these incorrect suspicions, although not permanent (if the suspected process is correct, the reply message will eventually be received), can happen infinitely often. This is due to the fact that the message delivery time could be greater than the fixed *default timeout* (see Fig. 2). In order to provide some useful accuracy, the timeout values must be augmented when processes are aware of having erroneously suspected a correct process. In this section, we present an extension to the basic algorithm of Fig. 2, based on augmenting the timeout values, which satisfies the eventual weak accuracy property.

Eventual weak accuracy requires that, eventually, some correct process is never suspected by any correct process. In order to provide it, it is enough that this is satisfied for only one correct process. Our extension to the basic algorithm guarantees the existence of such a process, which we denote *leader*. Clearly, if we knew beforehand a correct process, eventual weak accuracy could be obtained by making all processes augment their timeout value with respect to this process each time they suspect it. This correct process would be *leader*. But since we cannot know in advance the correctness of any process, we need to devise another way to eventually have a correct and not-suspected process.

In our extension of the algorithm of Fig. 2, processes behave as follows. Initially, every process will consider a pre-agreed process (e.g. p_1) as an *initial candidate* to be *leader*. When a process that monitors this candidate suspects it, it considers its successor in the ring as new candidate and monitors it. This scheme is repeated every time the current candidate is suspected. (Note that a process not monitoring a candidate cannot suspect it.) If a process p stops suspecting a process q, previously considered as candidate, then p will augment its timeout value $\Delta_{p,q}$ [1]. If the previously suspected process q was not considered as candidate, then p will not change $\Delta_{p,q}$. This way, *leader* will be the first correct process in the ring starting from the *initial candidate* (inclusive). All processes monitoring it will eventually stop suspecting it, and processes that do not monitor it will never suspect it. This gives us the eventually weak accuracy property. Fig. 3 presents the extended algorithm in detail.

We now show that eventual weak accuracy holds with this algorithm, i.e., eventually some correct process is never suspected by any correct process.

Lemma 1. *After T_s, any correct process p will suspect leader for no more than Δ_{rtt} time, each time it does.*

Proof. Remember that, after T_s, Δ_{rtt} is a bound on the monitoring round-trip time. A correct process p suspects *leader* after sending an ARE-YOU-ALIVE? message to it at time t and not receiving an I-AM-ALIVE reply message in $\Delta_{p,leader}(t)$

[1] For simplicity of the algorithm, instead of increasing timeouts when we stop suspecting, we increase timeouts as soon as we suspect a candidate.

Every process p executes:

```
$initial\_cand_p \leftarrow$ pre-agreed process
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task 1:
  loop
    wait($mutex_p$)
    send ARE-YOU-ALIVE? to $target_p$
    $t_{out} \leftarrow \Delta_{p,target_p}$
    $received \leftarrow false$
    signal($mutex_p$)
    delay $t_{out}$
    wait($mutex_p$)
    if not $received$
      if $initial\_cand_p \in \{succ(p), \ldots, target_p\}$
        $\Delta_{p,target_p} \leftarrow \Delta_{p,target_p} + 1$
      $L_p \leftarrow L_p \cup \{target_p\}$
      $target_p \leftarrow succ(target_p)$
    end if
    signal($mutex_p$)
  end loop

|| Task 2:
    ...          {Same as algorithm in Fig. 2}
coend
```

Fig. 3. Extension to the algorithm of Fig. 2 to provide eventual weak accuracy.

time. Since, by definition, *leader* is a correct process, the I-AM-ALIVE message will arrive at most at time $t + \Delta_{rtt}$ ($T_s + \Delta_{rtt}$ if $t < T_s$). At this moment *leader* is removed from L_p, the list of suspected processes of p.

Lemma 2. *Any correct process p will suspect leader a finite number of times.*

Proof. Let p be some correct process. Since the value of T_s is finite, p suspects *leader* a finite number of times before T_s. After that, from the algorithm, each time p suspects *leader*, the value of $\Delta_{p,leader}$ is incremented by one. After suspecting *leader* a finite number of times, $\Delta_{p,leader}$ will be greater than Δ_{rtt}. After this moment, p never suspects *leader* anymore.

Theorem 2. $\exists t_1 : \forall p \in correct, \forall t > t_1, leader \notin L_p(t)$

Proof. Let t_1^p be the instant at which a correct process p stops suspecting *leader* for the last time. (If p never suspects *leader*, $t_1^p = 0$.) Such an instant exists from Lemma 1 and Lemma 2. Then, after instant $t_1 = \max_{p \in correct}\{t_1^p\}$ no correct process p has *leader* in its list L_p.

Corollary 2. *The algorithm of Fig. 3 provides eventual weak accuracy.*

Observation 1 *The only difference between this algorithm and the algorithm of Fig. 2 is that in the former the values of $\Delta_{p,q}$ can change. Clearly, this does not affect the proof of Theorem 1. Hence, Corollary 1 also applies to this algorithm.*

Corollary 3. *The algorithm of Fig. 3 implements a failure detector of class $\diamond\mathcal{W}$.*

Proof. Follows from Corollary 2, Observation 1, and Corollary 1.

Every process p executes:

```
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task 1:
  loop
    wait($mutex_p$)
    send ARE-YOU-ALIVE? to $target_p$
    $t_{out} \leftarrow \Delta_{p,target_p}$
    $received \leftarrow false$
    signal($mutex_p$)
    delay $t_{out}$
    wait($mutex_p$)
    if not $received$
      $\Delta_{p,target_p} \leftarrow \Delta_{p,target_p} + 1$
      $L_p \leftarrow L_p \cup \{target_p\}$
      $target_p \leftarrow succ(target_p)$
    end if
    signal($mutex_p$)
  end loop

|| Task 2:
    ...          {Same as algorithm in Fig. 2}
coend
```

Fig. 4. Extension to the algorithm of Fig. 2 to provide eventual strong accuracy.

5 Extending the Basic Algorithm to Provide Eventual Strong Accuracy

Eventual strong accuracy requires that, eventually, no correct process is ever suspected by any correct process. In this section, we propose another extension to the basic algorithm of Section 3 which satisfies this property. Broadly, the extension consists in each process augmenting its timeout values with respect to all processes it incorrectly suspects. This way every process will augment the timeout value with respect to its closest correct successor in the ring, and will thus eventually stop suspecting it (and hence, any other correct process). This gives us the eventually strong accuracy property. Fig. 4 presents the extended algorithm in detail.

We now show that eventual strong accuracy holds with the algorithm in Fig. 4. We start with two lemmas, whose proofs are similar to those of Lemma 1 and Lemma 2, respectively, and are omitted.

Lemma 3. *After T_s, any correct process p will suspect $corr\_succ(p)$ for no more than Δ_{rtt} time, each time it does.*

Lemma 4. *Any correct process p will suspect $corr\_succ(p)$ a finite number of times.*

Theorem 3. $\exists t_2 : \forall p \in correct, \forall q \in correct, \forall t > t_2, q \notin L_p(t)$

Proof. Let t_2^p be the instant at which a correct process p stops suspecting $corr\_succ(p)$ for the last time. (If p never suspects $corr\_succ(p)$, $t_2^p = 0$.) Such an instant exists from Lemma 3 and Lemma 4. Then, after instant $t_2 = \max_{p \in correct} \{t_2^p\}$

no correct process p has $corr\_succ(p)$ in its list L_p. Then, after t_2, each correct process p only suspects processes in $succ(p), \ldots, pred(corr\_succ(p))$, which are not correct by the definition of $corr\_succ(p)$. Therefore, no correct process q is in L_p after t_2.

Corollary 4. *The algorithm of Fig. 4 provides eventual strong accuracy.*

Note that Observation 1 still applies to this algorithm. Hence, the following corollary, that follows from Corollary 4, Observation 1, and Corollary 1.

Corollary 5. *The algorithm of Fig. 4 implements a failure detector of class* $\diamond\mathcal{Q}$.

6 Extending the Previous Algorithms to Provide Strong Completeness

In this section we present an extension to the previous algorithms to provide strong completeness, while preserving accuracy. By combining this extension with the algorithms that implement failure detectors of classes $\diamond\mathcal{W}$ and $\diamond\mathcal{Q}$, presented in previous sections, we obtain implementations of failure detectors of classes $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$, respectively.

Strong completeness requires that, eventually, every process that crashes is permanently suspected by every correct process. In [2], Chandra and Toueg presented a distributed algorithm that transforms weak completeness into strong completeness. Broadly, in their algorithm, every process periodically *broadcasts* (sends to every other process) its *local* list of suspected processes. Upon reception of these lists, each process builds a *global* list of suspected processes, which provides strong completeness. Clearly, in this algorithm each correct process periodically sends n messages, with the total number of messages exchanged being at least $n\mathcal{C}$.

In our extension, we follow a similar approach. Besides its local list L_p of suspected processes, each process p has a global list G_p of suspected processes. While L_p only holds the suspected processes between p and the process p is monitoring ($target_p$), G_p holds all the processes that are being suspected in the system. Now, the global lists are the ones providing strong completeness.

In order to correctly build the global lists, processes need to propagate their local lists. However, instead of periodically broadcasting its local list, every process will only send its global list (which contains the local list) to the process it is monitoring. This process, upon reception of that list, updates its global list and further propagates it. Note that, since we use the ring arrangement of processes, each process at most sends and receives one message periodically, and the total number of messages exchanged is $O(n)$ in the worst case, which eventually becomes $O(\mathcal{C})$. Furthermore, instead of using specific messages to send the global lists, we can piggyback the global lists in the ARE-YOU-ALIVE? messages inherent to the monitoring action. This way, there is no increment in message exchanges

```
Every process $p$ executes:

{if the algorithm needs it:
$initial\_cand_p \leftarrow$ pre-agreed process}
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$G_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task 1:
  loop
    wait($mutex_p$)
    send ARE-YOU-ALIVE?
        —with $G_p$— to $target_p$
    $t_{out} \leftarrow \Delta_{p,target_p}$
    $received \leftarrow false$
    signal($mutex_p$)
    delay $t_{out}$
    wait($mutex_p$)
    if not $received$
      {Update $\Delta_{p,target_p}$ if required}
      $G_p \leftarrow G_p \cup \{target_p\}$
      $L_p \leftarrow L_p \cup \{target_p\}$
      $target_p \leftarrow succ(target_p)$
    end if
    signal($mutex_p$)
  end loop
|| Task 2:
  loop
    receive message $m$ from a process $q$
    wait($mutex_p$)
    case
      $m$ = ARE-YOU-ALIVE? —with $G_q$—:
        send I-AM-ALIVE to $q$
        if $q \in L_p$
          $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
          $target_p \leftarrow q$
          $received \leftarrow true$
        end if
        $G_p \leftarrow G_q \cup L_p - \{p, q\}$
      $m$ = I-AM-ALIVE:
        case
          $q = target_p$:
            $received \leftarrow true$
          $q \in L_p$:
            $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
            $G_p \leftarrow G_p - \{q\}$
            $target_p \leftarrow q$
            $received \leftarrow true$
          else discard $m$
        end case
    end case
    signal($mutex_p$)
  end loop
coend
```

Fig. 5. Extension to the previous algorithms to provide strong completeness.

from the previous algorithms. Fig. 5 presents the extended algorithm in detail. We now show that strong completeness holds, while accuracy is preserved, with this algorithm.

Observation 2 *The only difference between this algorithm and the previous ones is the handling of the global lists of suspected processes G_p, while the local lists L_p are handled as before. Hence, Theorem 1 and whichever corresponds of Theorem 2 and Theorem 3 are still applicable to this algorithm.*

Observation 3 $\forall p \in \Pi, \forall t, L_p(t) \subseteq G_p(t)$.

Observation 4 $\forall p \in$ *correct*, $\forall t$, *p will eventually receive* ARE-YOU-ALIVE? *messages after t.*

Lemma 5. $\exists t_3 : \forall q \in crashed, \forall p \in correct, \forall t \geq t_3, q \in G_p(t)$.

Proof. Let us assume we are at least at instant t_0 as defined in Theorem 1. We know that at this instant any process $q \in crashed$ has already failed and has been permanently included in $L_{corr\_pred(q)}$.

Let us assume now that we have a process $q \in crashed$ and a process $p \in correct$. We claim that q will eventually be permanently included in G_p. We use strong induction on the number of correct processes in the set $\{corr\_pred(q), \ldots, p\}$. For the base case we assume there is only one correct process in the set, i.e., $p = corr\_pred(q)$. Hence, from Theorem 1, q is permanently in L_p and, from Observation 3, q will be permanently in G_p in this case.

We will now prove that, if the claim holds for any number $1 \leq c \leq i-1$ of correct processes in the set $\{corr\_pred(q), \ldots, p\}$, it also holds when the number of correct processes in the set is i. To do so, we show first that there is a time t' after which p receives ARE-YOU-ALIVE? messages and all of them carry global lists containing q. From that, it is immediate to see in the algorithm that, after receiving the first such ARE-YOU-ALIVE? message, q will be permanently included in G_p. Let us assume the number of correct processes in the set $\{corr\_pred(q), \ldots, p\}$ be $i > 1$. By induction hypothesis, there is a time t'' at which any correct process $r \in \{corr\_pred(q), \ldots, corr\_pred(p)\}$ permanently contains q in its global list G_r. Also, there is a time $t' = \max(t'', T_s) + \Delta_{msg}$ at which all the ARE-YOU-ALIVE? messages sent to p before t'' have been received. From Observation 4, process p will receive new ARE-YOU-ALIVE? messages after t'. Let be an ARE-YOU-ALIVE? message received by p from a process s at a time $t > t'$. There are two cases to consider:

- $s \in \{corr\_pred(q), \ldots, corr\_pred(p)\}$. In this case, from the induction hypothesis and the definition of t', we know that the global list G_s carried by the ARE-YOU-ALIVE? message contains q.
- $s \in \{p, \ldots, corr\_pred(corr\_pred(q))\}$. In this case, it can be seen from the algorithm that if p receives an ARE-YOU-ALIVE? message from s, then necessarily, at the time of sending the message, $p = target_s$ and L_s contained q. Therefore, from Observation 3, the G_s carried by the ARE-YOU-ALIVE? message contains q.

The following lemma states that the algorithm of Fig. 5 preserves eventual accuracy.

Lemma 6. *Let p be any correct process. If there is a time after which no correct process q contains p in L_q, then there is a time after which no correct process q contains p in G_q.*

Proof. Let us assume we are at least at instant t_0 as defined in Theorem 1. We know that at this instant any process in *crashed* has already failed. Let p be a correct process and $t''' \geq t_0$ be an instant such that $\forall t \geq t''', \forall q \in correct, p \notin L_q$.

Let us assume now that we have a process $q \in correct$. We claim that there is a time after which p is never in G_q. We use strong induction on the number of correct processes in the set $\{p, \ldots, q\}$. For the base case, we assume there is only one correct process in the set, i.e., $p = q$. It is easy to observe from the algorithm that p will never include itself in G_p.

We will now prove that, if the claim holds for any number $1 \leq c \leq i - 1$ of correct processes in the set $\{p, \ldots, q\}$, it also holds when the number of correct processes in the set is i. To do so, we show first that there is a time t' after which q receives ARE-YOU-ALIVE? messages and all of them carry global lists not containing p. From that, it is immediate to see in the algorithm that, after receiving the first such ARE-YOU-ALIVE? message, p will be removed (if needed) and never included again in G_q. Let us assume the number of correct processes in the set $\{p, \ldots, q\}$ be $i > 1$. By induction hypothesis, there is a time t'' after which any correct process $r \in \{p, \ldots, \mathit{corr\_pred}(q)\}$ does not contain p in its global list G_r. Also, there is a time $t' = \max(t'', T_s) + \Delta_{msg}$ at which all the ARE-YOU-ALIVE? messages sent to q before t'' have been received. From Observation 4, process q will receive new ARE-YOU-ALIVE? messages after t'. Let be an ARE-YOU-ALIVE? message received by q from a process s at a time $t > t'$. There are two cases to consider:

- $s \in \{p, \ldots, \mathit{corr\_pred}(q)\}$. In this case, from the induction hypothesis and the definition of t', we know that the global list G_s carried by the ARE-YOU-ALIVE? message does not contain p.
- $s \in \{q, \ldots, \mathit{corr\_pred}(p)\}$. This case cannot happen, because it would imply that p is in the local list L_s.

Combining both lemmas it is immediate to derive the following theorem.

Theorem 4. *The algorithm of Fig. 5 provides strong completeness while preserving accuracy.*

Corollary 6. *The algorithm of Fig. 5, combined with the algorithm of Fig. 3 or Fig. 4, implements failure detectors of classes $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$, respectively.*

7 Performance Analysis

In this section, we will evaluate the performance of the presented algorithms in terms of the number and size of the exchanged messages. Observe that failure detection is an on-going activity that inherently requires an infinite number of messages. Furthermore, the pattern of message exchange between processes can vary over time (and need not be periodic), and different algorithms can have completely different patterns. For these reasons, we have to make some assumptions in order to use the number of messages as a meaningful performance measure. We will first assume that the algorithms execute in a periodic fashion, so that we can count the number of messages exchanged in a period. Secondly, to be able to compare the number of messages exchanged by different algorithms, we must assume that their respective periods have the same length.

Under the above assumptions, in our algorithms each correct process periodically polls only one other process. Each polling involves two messages. Thus, a total of no more than $2n$ messages would be periodically exchanged. Eventually, this amount becomes $2\mathcal{C}$, since there will be only $\mathcal{C}$ correct processes remaining in the system. This compares favorably with Chandra and Toueg's algorithm, which requires a periodic exchange of at least $n\mathcal{C}$ messages.

Concerning the size of the messages, our algorithms implementing failure detectors with weak completeness ($\diamond\mathcal{W}$ and $\diamond\mathcal{Q}$) require messages of $\Theta(\log n)$ bits (to identify the sender). On the other hand, the algorithms implementing failure detectors with strong completeness ($\diamond\mathcal{S}$ and $\diamond\mathcal{P}$) require messages of $\Theta(n)$ bits, since we can code the global list G_p of suspected processes in n bits (one bit per process).

Chandra and Toueg's algorithm, which only implements $\diamond\mathcal{P}$, requires messages of $\Theta(\log n)$ bits. This size is smaller than the size needed by our $\diamond\mathcal{P}$ algorithm. However, the total amount of information periodically exchanged in our algorithm is $\Theta(n^2)$ bits, while in theirs it is $\Theta(n^2 \log n)$ bits. Furthermore, each message that is sent involves a fixed overhead. In this sense, our algorithm presents an edge, since it involves less messages.

8 Conclusions and Future Work

In this paper we have proposed several algorithms to implement failure detectors of classes $\diamond\mathcal{W}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$. These algorithms are efficient alternatives to the algorithm implementing $\diamond\mathcal{P}$ proposed by Chandra and Toueg [2].

Apparently, the time to propagate the failure information in our algorithms is larger than the time in Chandra and Toueg's algorithm. We need to further study this performance parameter both theoretically and empirically.

As pointed out in the previous section, the number of messages periodically exchanged is not a general enough performance measure, since algorithms need not be periodic. We are studying new ways of evaluating the performance of this kind of algorithms.

References

1. M. Aguilera and S. Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)*, LNCS, Springer-Verlag, Germany, Sep. 1997. 34
2. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43 (2), pages 225-267, Mar. 1996. 34, 34, 34, 35, 35, 35, 35, 36, 36, 36, 38, 44, 48
3. T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43 (4), pages 685-722, Jul. 1996. 36
4. D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34 (1), pages 77-97, Jan. 1987. 35

5. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35 (2), pages 288-323, Apr. 1988. 35, 35, 35
6. C. Fetzer and F. Cristian. Fail-Aware Failure Detectors. *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS)*, Canada, Oct. 1996. 34
7. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32 (2), pages 374-382, Apr. 1985. 34, 35
8. R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. *Proceedings of the 14th Symposium on Reliable Distributed Systems (SRDS)*, Germany, Sep. 1996. 35
9. R. Guerraoui and A. Schiper. Gamma-Accurate Failure Detectors. *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*, LNCS, Springer-Verlag, Italy, Oct. 1996. 34
10. M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27 (2), pages 228-234, Apr. 1980. 34