Optimal Implementation of the Weakest Failure Detector for Solving Consensus *

Mikel Larrea
Universidad Pública de Navarra
31006 Pamplona, Spain
mikel.larrea@unavarra.es

Antonio Fernández
Universidad Rey Juan Carlos
28933 Móstoles, Spain
afernandez@acm.org

Sergio Arévalo
Universidad Rey Juan Carlos
28933 Móstoles, Spain
s.arevalo@escet.urjc.es

Abstract

The concept of unreliable failure detector *was introduced by Chandra and Toueg [2] as a mechanism that provides information about process failures. Depending on the properties the failure detectors guarantee, they proposed a taxonomy of failure detectors. It has been shown that one of the classes of this taxonomy, namely* Eventually Strong *($\Diamond\mathcal{S}$), is the weakest class allowing to solve the Consensus problem.*

In this paper, we present a new algorithm implementing $\Diamond\mathcal{S}$. Our algorithm guarantees that eventually all the correct processes agree on a common correct process. This property trivially allows us to provide the accuracy and completeness properties required by $\Diamond\mathcal{S}$. We show, then, that our algorithm is better than any other proposed implementation of $\Diamond\mathcal{S}$ in terms of the number of messages and the total amount of information periodically sent. In particular, previous algorithms require to periodically exchange at least a quadratic amount of information, while ours only requires $O(n \log n)$ (where n is the number of processes).

However, we also propose a new measure to evaluate the efficiency of this kind of algorithms, the eventual monitoring degree, *which does not rely on a periodic behavior and expresses better the degree of processing required by the algorithms. We show that the runs of our algorithm have optimal eventual monitoring degree.*

1. Introduction

The concept of *unreliable failure detector* was introduced by Chandra and Toueg in [2]. They 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. [4].) They also showed in [1] that one of the classes of failure detectors they defined, namely *Eventually Strong* ($\Diamond\mathcal{S}$), is the weakest class allowing to solve Consensus[1].

Since then, many distributed fault-tolerant algorithms have been designed based on Chandra-Toueg's unreliable failure detectors [5, 6, 9, 11]. Almost all of them consider a system model in which the failure detector they require is available, i.e., an asynchronous system augmented with a failure detector, such that the algorithm is designed on top of it. In this work we have taken a different approach, investigating how to implement those unreliable failure detectors.

From the results of Fischer et al. and those of Chandra and Toueg, it can be derived the impossibility of implementing failure detectors strong enough to solve the distributed Consensus problem in a pure asynchronous system. In [2], Chandra and Toueg presented a timeout-based algorithm implementing an *Eventually Perfect* ($\Diamond\mathcal{P}$) failure detector —a class strictly stronger than $\Diamond\mathcal{S}$— in models of partial synchrony [3]. This algorithm is based on all-to-all communication: each process periodically sends an I-AM-ALIVE message to all processes, in order to inform them that it has not crashed, and thus requires a quadratic number of messages to be periodically sent. More recently, Larrea et al. [7] proposed more efficient algorithms implementing several classes of failure detectors, including $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$. These algorithms are based on a ring arrangement of the processes, and require only a linear number of messages to be periodically sent.

1.1. Unreliable Failure Detectors

An *unreliable failure detector* is a mechanism that provides (possibly incorrect) information about faulty processes. When it is queried, the failure detector returns a list of processes believed to have crashed (suspected processes). In [2], failure detectors were characterized in terms of two properties: *completeness* and *accuracy*. Complete-

*Research partially supported by the Spanish Research Council (CICYT), under contract numbers TIC99-0280-C02-02, TEL99-0582, and TIC98-1032-C03-01, and the Madrid Regional Research Council (CAM), under contract number CAM-07T/00112/1998.

[1] In fact, the *Eventually Weak* failure detector class, $\Diamond\mathcal{W}$, is presented as the weakest one for solving Consensus. However, Chandra and Toueg have shown in [2] that $\Diamond\mathcal{S}$ and $\Diamond\mathcal{W}$ are equivalent.

ness 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 [2], which combined yield eight classes of failure detectors.

In this paper, we focus on the following completeness and accuracy properties, from those defined in [2]:

- *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 the rest of processes in the system. Such a failure detector is clearly useless, since it provides no information about failures.

- *Eventual Strong Accuracy.* There is a time after which no correct processes is 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 Figure 1. As we said, Chandra et al. showed in [1] that $\diamond\mathcal{W}$ is the weakest class of failure detectors required for solving the Consensus problem, and in [2] that $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent. For this reason we say in this paper that $\diamond\mathcal{S}$ is the weakest class of failure detectors for solving Consensus.

It is worth noting here that the equivalence of $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ does not come for free, i.e., not all failure detectors in $\diamond\mathcal{W}$ are in $\diamond\mathcal{S}$. Instead, it means that any failure detector in $\diamond\mathcal{W}$ can be extended with a simple distributed algorithm to obtain a failure detector in $\diamond\mathcal{S}$. Since all the Consensus algorithms we know of require at least a failure detector of class $\diamond\mathcal{S}$ (e.g. [2, 6, 9, 11]), if the efficiencies of $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ failure detectors are similar, it is more efficient to implement a failure detector of class $\diamond\mathcal{S}$ instead of implementing one of class $\diamond\mathcal{W}$ and running the extension algorithm on top of it[2].

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

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

1.2. Our Results

In this paper, we present a new algorithm implementing $\diamond\mathcal{S}$. Our algorithm guarantees that, eventually, all the correct processes agree on a common correct process. This property allows us to provide the accuracy required by $\diamond\mathcal{S}$. Then, by suspecting all the other processes, we trivially obtain the completeness required by $\diamond\mathcal{S}$.

We have compared our algorithm with the implementations of $\diamond\mathcal{S}$ proposed by Chandra and Toueg [2] and by Larrea et al. [7] in terms of the number and size of the messages periodically sent and the total amount of information periodically exchanged. We have found that, under the realistic assumption that the probability of process failure is low, our algorithm is clearly better in all the above performance measures.

In general, since algorithms implementing failure detectors need not necessarily be periodic, it is not always possible to evaluate their efficiency in terms of the amount of information periodically sent. For this reason, in order to evaluate the efficiency of this kind of algorithms independently of the communication pattern they use, we propose in this paper a new and (we believe) more adequate performance measure, which we call *eventual monitoring degree*. Informally, this measure counts the number of pairs of correct processes that will infinitely often communicate. We show that the runs of the algorithm presented in this paper are optimal, and at least as good as the runs of any algorithm implementing a failure detector of class $\diamond\mathcal{W}$, with respect to this measure.

It is well known that, in practice, every system goes through *stable* and *unstable* periods. Informally, a stable period is a period during which no process crashes and there are no timing failures, i.e., all messages arrive on time. On the other hand, an unstable period is a period during which processes may crash or there are timing failures, i.e., messages can be lost or arrive late. If we assume that the system is most frequently in stable periods, then the eventual monitoring degree is still a valid performance measure within each of these stable periods.

The rest of the paper is organized as follows. In Sec-

[2] The extension algorithm proposed in [2] requires a quadratic number of messages to be periodically exchanged, and any such algorithm will require at least a linear number of messages.

tion 2, we describe the system model and discuss different approaches in order to implement failure detectors, introducing the new measure to evaluate the efficiency of this kind of algorithms. In Section 3, we present our algorithm implementing a failure detector of class $\Diamond\mathcal{S}$. In Section 4, we prove the correctness of our algorithm. In Section 5, we evaluate the performance of our algorithm, and show that its runs are optimal with respect to the eventual monitoring degree measure. Finally, Section 6 concludes the paper.

2. The Model

2.1. System Model

We consider a distributed system consisting of a finite set Π of n processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving 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. Crashes are permanent, i.e., 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., $\mathcal{C} = |correct| > 0$.

We consider that processes are ordered. Without loss of generality, process p_i is preceded by processes $p_1, \ldots, p_{i-1}$, and followed by processes $p_{i+1}, \ldots, p_n$.

We consider the model of *partial synchrony* proposed by Chandra and Toueg in [2], which is a generalization of the models proposed by Dwork et al. in [3]. This model stipulates that, in every execution, there are bounds on process speeds and on message transmission times, but these bounds are not known *and* they hold only after some unknown time (called *GST* for *Global Stabilization Time*). In this model, we will denote by Δ_{msg} the maximum time, after GST, between the sending of a message and the delivery and processing by its destination process (assuming that both the sender and the destination have not failed). Clearly, Δ_{msg} depends on the existing bounds on process speeds and on message transmission times. 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. Upon request, each module provides its attached process with a list of processes it suspects to have crashed. These lists can differ from one module to another at a given time. We denote by $suspected_i$ the list of suspected processes of the failure detection module attached to process p_i. We assume that a process 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 use the term *process* instead of *failure detection module*. It will be clear from the context if we are referring to the failure detection module or the process attached to it. We consider that a process cannot crash independently of its attached failure detection module.

Any algorithm implementing a failure detector requires that some processes detect whether other processes have crashed, and take proper action if so. There are mainly two possible ways to implement this failure detection: the *push* model and the *pull* model. In the push model, processes are permanently sending I-AM-ALIVE messages to the processes in charge of detecting their potential failure. In the pull model, the later ask the former for such messages. In any case, the only way a process can show it has not crashed is by sending messages to other processes.

The algorithm presented in this paper is based on the push model. At any time, at least one process is sending I-AM-ALIVE messages periodically to a subset of the processes in the system. Processes monitor each other by waiting for these periodical I-AM-ALIVE messages. To monitor a process p_j, process p_i uses an estimated value –timeout– that tells how much time it has to wait for the I-AM-ALIVE message from p_j. This time value is denoted by $\Delta_{i,j}$. Then, if after $\Delta_{i,j}$ time p_i did not receive the I-AM-ALIVE message from p_j, it suspects that p_j has crashed. We need to allow these time values to vary over time in our algorithm. We use $\Delta_{i,j}(t)$ to denote the value of $\Delta_{i,j}$ at time t.

It must be clear that, in general, different algorithms can use a different period to exchange information between processes. Furthermore, this exchange of information in a given algorithm need not be periodic. In this set up, it is difficult to use the number of messages periodically exchanged as the efficiency measure of this kind of algorithms. For this reason, on top of the number and size of messages periodically sent, we will also use the *eventual monitoring degree* measure to evaluate the efficiency of the algorithms, which we define as follows.

Definition 1 *The* eventual monitoring degree *of a run R of an algorithm A is the number of pairs (p_i, p_j) of correct processes, such that p_i sends infinitely many messages to p_j in run R.*

2.3. The Ω Failure Detector

In their proof of $\Diamond\mathcal{W}$ being the weakest failure detector for solving Consensus [1], Chandra et al. defined a new failure detector, denoted Ω.[3] The output of the failure detector module of Ω at a process p is a *single* process q, that p currently considers to be *correct*; we say that *p trusts q*. The failure detector Ω satisfies the following property:

- There is a time after which all the correct processes always trust the same correct process.

As with $\Diamond\mathcal{W}$, the output of the failure detector module of Ω at a process p may change with time, i.e., p may trust different processes at different times. Furthermore, at any given time t, two processes p and q may trust different processes.

It is straightforward to transform Ω into $\Diamond\mathcal{W}$ (and $\Diamond\mathcal{S}$). It can be done by forcing each process to suspect every process in the system except its trusted process. This gives us the completeness and accuracy properties required by $\Diamond\mathcal{W}$ (and $\Diamond\mathcal{S}$). As we will see, the algorithm presented in this paper follows this strategy, and actually implements Ω.

3. The Algorithm

In this section, we present our implementation of a failure detector of class $\Diamond\mathcal{S}$. In Figure 2, we present the algorithm that is the core of our $\Diamond\mathcal{S}$ failure detector implementation. This algorithm, when run by each process of the system, guarantees that eventually all the correct processes agree on a common correct process, denoted p_{leader}. This property trivially allows us to provide the eventual weak accuracy property required by $\Diamond\mathcal{S}$: eventually, p_{leader} is not suspected by any correct process. The strong completeness property of $\Diamond\mathcal{S}$ is reached by simply making every process p_i suspect all processes in the system except p_{leader}.

Each process p_i runs an instance of the algorithm of Figure 2, in which there is a local variable $trusted_i$. As we will show, eventually the value of $trusted_i$ for each correct process p_i will be the same, and $p_{trusted_i}$ will be the correct process p_{leader}. In fact, $trusted_i = leader$ will be the index of the first process $\in$ *correct* in the system (following the order $p_1, ..., p_n$). From the value of $trusted_i$, it can be derived the set *suspected$_i$*, which will satisfy the accuracy and completeness properties of $\Diamond\mathcal{S}$. Here we consider two possibilities, to make $suspected_i = \Pi - \{p_{trusted_i}\}$ or $suspected_i = \Pi - \{p_{trusted_i}, p_i\}$, depending on whether we want to preserve the intuitive notion of a process never suspecting itself.

```
Every process $p_i, i = 1, \ldots, n$ executes:

$trusted_i \leftarrow 1$
$\forall j \in \{1, \ldots, i-1\} : \Delta_{i,j} \leftarrow$ default timeout
$received_i \leftarrow false$

cobegin

|| Task 1:
  loop
    if $i = trusted_i$ then
      send I-AM-ALIVE to $p_{i+1}, \ldots, p_n$
    end if
    delay $\Delta_{HEARTBEAT}$
  end loop

|| Task 2:
  loop
    if $trusted_i < i$ then
      delay $\Delta_{i,trusted_i}$
      if $received_i$ then
        $received_i \leftarrow false$
      else
        $trusted_i \leftarrow trusted_i + 1$
      end if
    end if
  end loop

|| Task 3:
  loop
    receive I-AM-ALIVE from $p_j$
    if $j = trusted_i$ then
      $received_i \leftarrow true$
    else if $j < trusted_i$ then
      $\Delta_{i,j} \leftarrow \Delta_{i,j} + 1$
      $trusted_i \leftarrow j$
      $received_i \leftarrow true$
    else
      discard message
    end if
  end loop

coend
```

Figure 2. Algorithm used to implement a failure detector of class $\Diamond\mathcal{S}$.

The algorithm of Figure 2 executes as follows. Initially, each process p_i starts with $trusted_i = 1$, which means that p_1 will be their first candidate to be the process p_{leader}. Process p_1 starts sending I-AM-ALIVE messages periodically to the rest of processes $p_2, \ldots, p_n$. In general, a process p_i will be sending I-AM-ALIVE messages periodically to its successors $p_{i+1}, \ldots, p_n$ if $i = trusted_i$ (Task 1). A process p_i such that $trusted_i \neq i$, just waits for periodical I-AM-ALIVE messages from the process $p_{trusted_i}$. If it does not receive an I-AM-ALIVE message on time (within some timeout period $\Delta_{i,trusted_i}$), then it suspects that $p_{trusted_i}$ has crashed and chooses the next candidate to be the process p_{leader} by increasing $trusted_i$ by one (Task 2).

If, later on, a process p_i receives an I-AM-ALIVE message from a process p_j, such that $j < trusted_i$, then p_i will

[3] Actually, 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 $\Diamond\mathcal{W}$).

stop assuming that p_j has crashed, and will trust p_j again (by making $trusted_i = j$). In order to prevent this from happening an infinite number of times, p_i also increases the value of the timeout period $\Delta_{i,j}$ (Task 3). Moreover, if p_i was sending I-AM-ALIVE messages periodically, it will automatically stop sending them, since now $i \neq trusted_i$.

4. Correctness Proof

In this section, we show that the algorithm of Figure 2, combined with either of the proposed definitions of $suspected_i$ ($\Pi - \{p_{trusted_i}\}$ or $\Pi - \{p_{trusted_i}, p_i\}$), implements a failure detector of class $\diamond\mathcal{S}$. Among the processes $p_1, \ldots, p_n$, let p_{leader} be the correct process with the smallest index. The key of the proof is to show that, eventually and permanently, $trusted_i = leader$ for every correct process p_i. Thus, with either definition of $suspected_i$, eventually some correct process (namely p_{leader}) is never suspected by any correct process, which provides the eventual weak accuracy property required by $\diamond\mathcal{S}$, and eventually all crashed processes are permanently suspected by all correct processes, which provides the strong completeness property required by $\diamond\mathcal{S}$.

All time instants considered in the rest of this section are assumed to be after GST (Global Stabilization Time). We also assume that, at these instants, all messages sent before GST have already been delivered and processed. These assumptions allow us to consider in the rest of the section that the unknown bounds on process speeds and on message transmission times hold. We denote by $trusted_i(t)$ the value of $trusted_i$ at time t.

Lemma 1 *$\exists t_0 : \forall t > t_0, \forall p_i \in correct, trusted_i(t) \geq leader$.*

Proof: Let p_i be any correct process. By definition of p_{leader}, eventually all its predecessors, $p_1, \ldots, p_{leader-1}$, will crash. Consider a time t' at which all the predecessors of p_{leader} have crashed and all their messages have already been delivered and processed (in Task 3). Then, if $trusted_i(t') < leader$, eventually (at most at time $t'' = t' + \Delta_{i,trusted_i}(t') +$ the time to execute the corresponding instructions in Task 2) the variable $received_i$ will take the value $false$ (see Task 2). By Task 3, $received_i$ will not become $true$ until $trusted_i \geq leader$. Since at time t' all the predecessors of p_{leader} have crashed and by the algorithm, eventually (at most at time $t_i = t'' + \sum_{j=1}^{leader-1} \Delta_{i,j}(t') +$ the time to execute the corresponding instructions in Task 2 $leader - 1$ times) the variable $trusted_i$ will take a value greater or equal to $leader$ (see Task 2). Since p_i will never receive after t_i any other message from processes $p_1, \ldots, p_{leader-1}$, the variable $trusted_i$ will never take a value below $leader$ (see Task 3).

Let $t_0 = \max\{t_i : p_i \in correct\}$. From the above reasoning, $\forall t > t_0, \forall p_i \in correct, trusted_i(t) \geq leader$. ■

Lemma 2 *$\forall t > t_0$, where t_0 is the same as in Lemma 1, $trusted_{leader}(t) = leader$.*

Proof: Note from Task 2 that $\forall t : trusted_{leader}(t) \leq leader$. Also, from Lemma 1, $\forall t > t_0$, $trusted_{leader} \geq leader$. Hence, $\forall t > t_0$, $trusted_{leader} = leader$. ■

Lemma 3 *After t_0, the process p_{leader} will be permanently sending* I-AM-ALIVE *messages periodically to all its successors $p_{leader+1}, \ldots, p_n$.*

Proof: It follows from Lemma 2 and Task 1. ■

Lemma 4 *Let $p_i \in correct : i \neq leader$. If at time $t' > t_0$, $trusted_i(t') > leader$, then $\exists t'' : t' < t'' \leq t' + \Delta_{HEARTBEAT} + \Delta_{msg}$ and $trusted_i(t'') = leader$.*

Proof: Note that, by definition of p_{leader}, p_i has to be a successor of p_{leader}. From Lemma 3, after time t_0 the process p_{leader} is permanently sending I-AM-ALIVE messages, with a period of $\Delta_{HEARTBEAT}$, to all its successors, including p_i. After t', the first I-AM-ALIVE message will be sent by p_{leader} at time $t' + \Delta_{HEARTBEAT}$ at the latest. Note that, since we have a partially synchronous system, this message takes a maximum time of Δ_{msg} to be delivered and processed by p_i. Hence, at some time $t'' \leq t' + \Delta_{HEARTBEAT} + \Delta_{msg}$, p_i will deliver and process an I-AM-ALIVE message from p_{leader}. From Lemma 1, $trusted_i \geq leader$ at t'', and then from Task 3, $trusted_i$ will take the value $leader$ at that time. ■

Lemma 5 *Let $p_i \in correct : i \neq leader$. After t_0, $trusted_i$ will change from leader to a value different from leader a finite number of times.*

Proof: Let us assume, by the way of contradiction, that $trusted_i$ changes from $leader$ to a value different from $leader$ an infinite number of times. From Lemma 4, the value of $trusted_i$ will be $leader$ at some time after t_0. From Task 2, $trusted_i$ changes from $leader$ to $leader + 1$ if two I-AM-ALIVE messages are received by p_i more than $\Delta_{i,leader}$ time apart. Note from Task 1 and from the fact that we have a partially synchronous system that two consecutive I-AM-ALIVE messages sent by p_{leader} are received and processed by p_i at most $\Delta_{HEARTBEAT} + \Delta_{msg}$ time apart. Also, from Lemma 4, the value of $trusted_i$ will become $leader$ again eventually.

Every time this happens, from Task 3, the value of $\Delta_{i,leader}$ is incremented by one. Hence, since this will happen an infinite number of times, eventually $\Delta_{i,leader}$ will be larger than $\Delta_{HEARTBEAT} + \Delta_{msg}$. However, after that happens $trusted_i$ will never change its value from *leader*, which is a contradiction. ■

Theorem 1 $\exists t_1 : \forall t > t_1, \forall p_i \in correct, trusted_i(t) = leader$.

Proof: Follows from Lemma 2 for the case $i = leader$, and from Lemma 4 and Lemma 5 for the case $i \neq leader$. ■

Corollary 1 *Let* $suspected_i$ *be defined as either* $\Pi - \{p_{trusted_i}\}$ *or* $\Pi - \{p_{trusted_i}, p_i\}$, $\forall p_i \in \Pi$. *The algorithm of Figure 2, combined with either of these definitions of* $suspected_i$*, implements a failure detector of class* $\Diamond\mathcal{S}$.

5. Performance Analysis

In this section, we first evaluate the performance of the presented algorithm in terms of the number and size of the messages periodically sent. This evaluation shows that our algorithm compares favorably with the algorithms proposed by Chandra and Toueg [2] and by Larrea et al. [7]. After that, we evaluate our algorithm in terms of the eventual monitoring degree, as defined in Definition 1, which we consider more suitable for algorithms implementing failure detectors. We show that the runs of our algorithm are optimal with respect to this measure.

5.1. Number and size of messages and amount of information

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[4] fashion, so that we can count the number of messages sent in a period. Secondly, to be able to compare the number of messages sent by different algorithms, we must assume that their respective periods have the same length.

Algorithm	Nb. of msg.	Size of msg.	Amount of information	Eventual mon. deg.
CT	$n\mathcal{C}$	$\Theta(\log n)$	$\Theta(n\mathcal{C}\log n)$	$\mathcal{C}^2$
LAF	$2\mathcal{C}$	$\Theta(n)$	$\Theta(n\mathcal{C})$	$2\mathcal{C}$
Figure 2	$n-1$	$\Theta(\log n)$	$\Theta(n \log n)$	$\mathcal{C}-1$

Figure 3. Evaluation of different algorithms implementing failure detectors.

Under the above assumptions, the number of messages periodically sent in our algorithm is eventually at most $n-1$, since eventually only the process p_{leader} will be sending I-AM-ALIVE messages periodically to its successors. In Chandra-Toueg's algorithm implementing $\Diamond\mathcal{P}$, the number of messages periodically sent eventually becomes $n\mathcal{C}$, since eventually only the $\mathcal{C}$ correct processes will be sending messages periodically to all processes. Finally, in the algorithms implementing $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ of Larrea et al., the number of messages periodically sent is eventually $2\mathcal{C}$, since eventually each correct process communicates with its predecessor and its successor in the ring formed by the correct processes. Note that our algorithm requires less messages than Chandra-Toueg's. Compared with the algorithms of Larrea et al., if the probability of process failure is low, the value of $\mathcal{C}$ will be close to n and our algorithm will also require less messages.

However, note that only $\mathcal{C}-1$ messages out of the $n-1$ periodically sent by p_{leader} will be actually delivered and processed[5]. Similarly, the actual number of messages delivered and processed in Chandra-Toueg's algorithm is $\mathcal{C}^2$.

Concerning the size of the messages, our algorithm requires messages of $\Theta(\log n)$ bits, since each message needs to carry the identity of its sender. This is also the case for Chandra-Toueg's algorithm. However, the algorithms of Larrea et al. require messages of $\Theta(n)$ bits, since messages carry a list of suspected processes.

If we look now at the total amount of information periodically sent, our algorithm eventually requires $\Theta(n \log n)$ bits to be sent. The total amount of information periodically sent by Chandra-Toueg's algorithm is eventually $\Theta(n\mathcal{C}\log n)$ bits. Finally, the algorithms of Larrea et al. eventually require $\Theta(n\mathcal{C})$ bits to be sent.

5.2. Eventual monitoring degree

We now evaluate the algorithms in terms of the *eventual monitoring degree* of their runs, as defined in Definition 1. As pointed out in Sections 1 and 2, we consider this measure more adequate to evaluate the efficiency of algorithms

[4] For algorithms based on the push model, we assume that all I-AM-ALIVE messages are sent with the same period. For algorithms based on the pull model, we assume that there are no incorrect suspicions, and that all the timeout values are identical. Thus, the timeout value can be viewed as the periodicity of the algorithm.

[5] The amount of processing is better captured with the *eventual monitoring degree* measure, considered later in this section.

implementing failure detectors, since it is independent of the communication pattern they use, in particular of their potential periodicity. Moreover, in practice, this measure is close to the number of messages exchanged during a stable period.

It is easy to observe that the eventual monitoring degree of any run of the algorithm of Figure 2 is $\mathcal{C} - 1$, since eventually only the correct processes different from p_{leader} will receive messages, and they will receive them only from p_{leader}.

The eventual monitoring degree of any run of Chandra-Toueg's algorithm implementing $\diamond\mathcal{P}$ is $\mathcal{C}^2$, since eventually each correct process will receive messages from every correct process. Similarly, the eventual monitoring degree of any run of the algorithms implementing $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ of Larrea et al. is $2\mathcal{C}$, since eventually each correct process will only receive messages from its predecessor and its successor in the ring formed by the correct processes.

Figure 3 summarizes the evaluation of the different algorithms implementing failure detectors; CT denotes Chandra-Toueg's algorithm [2], and LAF denotes the ring-based algorithms of Larrea et al. [7].

We now show that any run of the algorithm of Figure 2 is optimal with respect to the eventual monitoring degree measure, i.e., any run of our algorithm has the minimum eventual monitoring degree needed to implement the weakest failure detector for solving Consensus. For generality, we will consider the class $\diamond\mathcal{W}$ of failure detectors, showing that the eventual monitoring degree required by a run of any algorithm implementing a failure detector of class $\diamond\mathcal{W}$ is at least $\mathcal{C} - k$, where k is the assumed minimum number of correct processes in the system. Observe that, by definition, $\mathcal{C} \geq k$. Note also that in Section 2 we assumed we have a system with $k = 1$. However, in other system models it is assumed a larger value of k (for instance, a majority of correct processes, i.e., $k > n/2$). We use the parameter k here for the generality of the lemma.

Lemma 6 *Any run R of an algorithm A implementing a failure detector of class $\diamond\mathcal{W}$ has an eventual monitoring degree of at least $\mathcal{C}_R - k$, where $\mathcal{C}_R$ is the number of correct processes in R, and k is the assumed minimum number of correct processes in the system.*

Proof: The claim trivially holds when $\mathcal{C}_R = k$. Let us assume now that $\mathcal{C}_R > k$. We will prove the claim by contradiction, showing that if R does not satisfy the claim (i.e., its eventual monitoring degree is less than $\mathcal{C}_R - k$), then we can find another run R' of algorithm A that violates some property of $\diamond\mathcal{W}$.

By the way of contradiction, let us assume that the eventual monitoring degree of the run R is at most $\mathcal{C}_R - k - 1$. Hence, there is some time t and there are at least $k + 1$ correct processes such that these processes do not receive messages from any other process after time t. From the fact that A implements a failure detector of class $\diamond\mathcal{W}$, the eventual weak accuracy property is satisfied in R. Therefore, there is a time t' and a correct process p_j such that p_j is never suspected by any correct process after t'.

Let us consider now another run R' of A, which behaves exactly the same as R up to a time instant $t'' > max(t, t')$. Let $p_1, \dots, p_k$ be k correct processes distinct from p_j which do not receive messages from any other process after time t. Let us assume that in R' all processes except $p_1, \dots, p_k$ crash at time t'', and $p_1, \dots, p_k$ are correct. Clearly, process p_j crashes in R' and, by the weak completeness property of A, p_j should be suspected by at least one of the correct processes $p_1, \dots, p_k$. Note that in R', after time t'' no message is received by $p_1, \dots, p_k$ from any other process, as it happened in R. Hence, from the point of view of processes $p_1, \dots, p_k$, both runs R and R' are indistinguishable. Therefore, processes $p_1, \dots, p_k$ take the same decisions in both runs. In particular, after time t' none of $p_1, \dots, p_k$ will suspect process p_j in R', as they do in R. This violates the weak completeness property in R' and, thus, A does not implement a failure detector of class $\diamond\mathcal{W}$, which is a contradiction. ■

Theorem 2 *For $k = 1$, where k is the assumed minimum number of correct processes in the system, any run of the algorithm of Figure 2 has an optimal eventual monitoring degree of $\mathcal{C} - 1$.*

Proof: Follows from Lemma 6 and the observation made at the beginning of this subsection. ■

6. Conclusions and Future Work

In this paper, we have presented a new algorithm implementing $\diamond\mathcal{S}$, the weakest failure detector for solving Consensus. Our algorithm compares favorably with the algorithms proposed by Chandra and Toueg [2] and by Larrea et al. [7] in terms of the number and size of the messages periodically sent. We have also proposed a more suitable measure to evaluate the efficiency of algorithms implementing failure detectors, which we call *eventual monitoring degree*, and have shown that the runs of the algorithm presented in this paper are optimal with respect to this measure.

Comparing to other algorithms, it may seem that our algorithm has a big loss of accuracy, because all processes except one are systematically suspected. In fact, it may sometimes work worse than others in algorithms like Chandra and Toueg's Consensus algorithm [2]. However, we believe this aspect, rather than being a problem, can be a benefit, since the fact that eventually all the lists of suspected processes are identical can be very helpful. In [8], we propose a Consensus algorithm that successfully exploits this property to solve Consensus more efficiently, i.e., in less rounds, than existing previous algorithms for $\diamond\mathcal{S}$.

In the algorithm of Figure 2, the timeout values of each process, Δ_{ij}, never decrease. So, in a network with possible transient communication failures, the Δ_{ij} values may eventually become too large. We are studying how these timeout values can be decreased. Clearly, this cannot be made independently of the applications using the failure detector. For instance, it would be possible to decrease the timeout values when no distributed algorithm using the failure detector, e.g. Consensus, is being executed.

Another line of future work is to improve the algorithm of Figure 2 for the case $k > 1$, where k is the assumed minimum number of correct processes in the system.

Acknowledgements

We are grateful to André Schiper for his valuable comments on earlier drafts of this paper.

References

[1] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

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

[5] R. Guerraoui, M. Larrea, , and A. Schiper. Non-blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th Symposium on Reliable Distributed Systems (SRDS'95)*, pages 41–51, September 1995.

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

[7] M. Larrea, S. Arévalo, and 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)*, pages 34–48. LNCS, Springer-Verlag, September 1999.

[8] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. Technical Report, Universidad Pública de Navarra, April 2000. Brief Announcement, 14th International Symposium on Distributed Computing (DISC'2000), Toledo, Spain.

[9] A. Mostefaoui and 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)*, pages 49–63. LNCS, Springer-Verlag, September 1999.

[10] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[11] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.