# Implementing the Weakest Failure Detector for Solving Consensus

Mikel Larrea
Universidad del País Vasco UPV/EHU
20018 San Sebastián, Spain
*mikel.larrea@ehu.es*

Antonio Fernández Anta
Institute IMDEA Networks
28918 Leganés, Spain
*antonio.fernandez@imdea.org*

Sergio Arévalo
Universidad Politécnica de Madrid
28031 Madrid, Spain
*sergio.arevalo@eui.upm.es*

## 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 several agreement problems, like Consensus. In this paper, algorithms that implement failure detectors in partially synchronous systems are presented. First two simple algorithms of the weakest class to solve Consensus, namely the Eventually Strong class ($\diamond S$), are presented. While the first algorithm is wait free, the second is $f$-resilient, where $f$ is a known upper bound on the number of faulty processes. Both algorithms guarantee that, eventually, all the correct processes agree permanently on a common correct process, i.e., they also implement a failure detector

---

Research partially supported by the Spanish Research Council, under grants TIN2005-09198-C02-01, TIN2007-67353-C02-02, and TIN2008-06735-C02-01, and the Comunidad de Madrid, under grant S-0505/TIC/0285.

A preliminary version of this article was presented at *SRDS'2000* [22].

of the class Omega ($\Omega$). They are also shown to be optimal in terms of the number of communication links used forever. Additionally, a wait-free algorithm that implements a failure detector of the Eventually Perfect class ($\Diamond P$) is presented. This algorithm is shown to be optimal in terms of the number of bidirectional links used forever.

*Keywords:* distributed computing, fault-tolerance, Consensus, failure detector, partial synchrony.

# 1 Introduction

The concept of unreliable failure detector was introduced by Chandra and Toueg in [7]. They showed how unreliable failure detectors can be used to solve the Consensus problem [35] in asynchronous systems. (This was shown to be impossible in a pure asynchronous system by Fischer et al. [14].) They also showed in [6] that one of the classes of failure detectors they defined, namely the Eventually Strong ($\Diamond S$) class, is the weakest allowing to solve Consensus in an asynchronous system with a majority of correct processes. In fact, the Eventually Weak failure detector class, denoted $\Diamond W$, is presented as the weakest one for solving Consensus. However, Chandra and Toueg have shown in [7] that $\Diamond S$ and $\Diamond W$ are equivalent in asynchronous systems with reliable channels. Since then, many fault-tolerant distributed algorithms have been designed based on Chandra-Toueg's unreliable failure detectors [15, 19, 31, 36]. 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. This work addresses a different problem, namely the implementation of these 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 Consensus problem in a pure asynchronous system. In [7], Chandra and Toueg presented a timeout-based algorithm implementing an Eventually Perfect ($\Diamond P$) failure detector —a class strictly stronger than $\Diamond S$— in models of partial synchrony [10]. 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. Also, a quadratic number of communication links are used forever. In [23], Larrea et al. propose more efficient algorithms implementing several classes of failure detectors, including $\diamond S$ and $\diamond P$. These algorithms are based on a ring arrangement of the processes, and require only a linear number of messages to be periodically sent. Consequently, only a linear number of communication links are used forever.

## 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 set of processes believed to have crashed (suspected processes). In [7], failure detectors were characterized in terms of two properties: *completeness* and *accuracy*. Completeness characterizes the failure detector capability of suspecting incorrect processes (processes that have actually crashed), while accuracy characterizes the failure detector capability of not suspecting correct processes. In this work, we focus on the following completeness and accuracy properties, from those defined in [7]:

- *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.

Note that, in isolation, completeness and accuracy are useless. For example, strong completeness can be satisfied by forcing every process to permanently suspect every other process in the system. Similarly, eventual strong accuracy can be satisfied by forcing every process to never suspect any process in the system. Such failure detectors are clearly useless, since they provide no information about failures. To be useful, a failure detector must satisfy some completeness *and* some accuracy.

Combining in pairs these completeness and accuracy properties, four different failure detector classes are obtained, which are presented in Figure 1. As previously said, Chandra

| | Eventual Strong Accuracy | Eventual Weak Accuracy |
|---|---|---|
| Strong Completeness | *Eventually Perfect*<br>$\diamond P$ | *Eventually Strong*<br>$\diamond S$ |
| Weak Completeness | *Eventually Quasi-Perfect*<br>$\diamond Q$ | *Eventually Weak*<br>$\diamond W$ |

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

et al. showed in [6] that $\diamond W$ is the weakest class of failure detectors required for solving the Consensus problem in an asynchronous system with a majority of correct processes, and in [7] that classes $\diamond S$ and $\diamond W$ are equivalent. For this reason it is said that $\diamond S$ is the weakest class of failure detectors for solving Consensus.

It is worth noting here that the equivalence of $\diamond S$ and $\diamond W$ does not come for free, i.e., not all failure detectors in $\diamond W$ are in $\diamond S$. Instead, it means that any failure detector in $\diamond W$ can be extended with a simple distributed algorithm to obtain a failure detector in $\diamond S$. Since most Consensus algorithms proposed require at least a failure detector of class $\diamond S$ (e.g., [7, 19, 31, 36]), if the costs of implementing $\diamond S$ and $\diamond W$ failure detectors are similar, it is more efficient to directly implement a failure detector of class $\diamond S$, instead of implementing one of class $\diamond W$ and running the extension algorithm on top of it. For example, the extension algorithm proposed in [7] requires a quadratic number of messages to be periodically exchanged.

### The Omega Failure Detector

In their proof of $\diamond W$ being the weakest class of failure detectors for solving Consensus [6], Chandra et al. defined a new failure detector class, called Omega ($\Omega$). To prove their result, Chandra et al. show first that $\Omega$ is at least as strong as $\diamond W$, and then that any failure detector $D$ that can be used to solve Consensus is at least as strong as $\Omega$, and hence at least as strong as $\diamond W$. The output of the failure detection module of $\Omega$ at a process $p$ is a *single* process $q$, that $p$ currently considers to be correct: $p$ *trusts* $q$. A failure detector in $\Omega$ satisfies the following property:

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

It is said that $\Omega$ provides an eventual leader election functionality.

As with $\Diamond W$, the output of the failure detection module of a detector in $\Omega$ 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. However, note that the period during which the output of $\Omega$ is arbitrary is finite.

It is straightforward to transform a detector in $\Omega$ into one in $\Diamond W$ (and $\Diamond S$) at no additional communication cost if the system membership is known to all processes (otherwise, even $\Diamond W$ cannot be implemented [21]). 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 $\Diamond W$ (and $\Diamond S$). As we will see, the $\Diamond S$ algorithms presented in this paper follow this strategy.

Observe that while $\Omega$ can be transformed into $\Diamond W$ and $\Diamond S$ without any communication, transforming $\Diamond W$ or $\Diamond S$ into Omega is far from being trivial and requires communication [9, 30]. Therefore, a lower bound result for $\Diamond S$ directly implies a lower bound result for $\Omega$, while the opposite direction is not true.

## 1.2 Related Work

In the latest years several authors have investigated the implementation of failure detectors. A lot of this effort has gone to provide implementations of (a detector in) Omega in the weakest possible system. (We will often use the name of the failure detector class to denote a detector of the class. Whether we mean a detector or the whole class shall be clear from the context.) In [1], Aguilera et al. introduce the notion of stable leader election, and propose several algorithms implementing Omega in a system where all links to and from some correct process are eventually timely. (A link is eventually timely if there is a time $GST$ and a bound $\delta$ such that, after $GST$, all messages sent on the link are received in $\delta$ time.) In [2, 4], they propose an algorithm implementing Omega in a system where only the output links of an unknown correct process are eventually timely, but in which a quadratic number of links must carry messages forever. With the additional assumption that some unknown correct process has all its input and output links fair, Aguilera et al. propose an algorithm such that eventually only one process, e.g., the leader, sends messages. More recently, they study in [3] the degree of synchrony required to implement Omega when the maximum number of processes that can crash is known. There are other recent papers that also use some form of eventual timeliness as the system property required to implement $\Omega$ [11, 20, 21, 26].

In [27], Mostéfaoui et al. propose a new look at the implementation of $\Omega$ failure detectors, based on the pattern of message arrivals instead of their timing. The proposed approach is based on a query/response mechanism and assumes that the query/response messages exchanged obey a pattern where the responses from some processes to a query arrive among the first ones. This approach is used in [33] to implement Omega. Furthermore, they show in [28, 29] that this new approach can be advantageously combined with the classical approach based on partial synchrony assumptions to implement failure detectors with eventual accuracy using hybrid protocols. Timing and pattern assumptions have been combined to implement $\Omega$ in [12, 34].

Another line of research has to do with implementing failure detectors with probabilistic guarantees. Chen et al. study in [8] the quality of service of failure detectors. In [5], Bertier et al. propose a new probabilistic implementation of a failure detector. This implementation is a variant of the heartbeat failure detector of [8] which is adaptable and can support scalable applications. In [13], Fetzer et al. propose a failure detection protocol that relies as much as possible on application messages to monitor the processes, using control messages only when no application messages are sent by the monitoring process to the observed process. In [16], Gupta et al. look at quantifying the optimal network load (in messages per second, with messages having a size limit) of failure detectors as a function of two application-specified requirements, (1) quick failure detection, and (2) accuracy of failure detection. In [18], Hayashibara et al. present a novel approach to adaptive failure detectors, called $\varphi$-failure detectors, which dynamically adapt to application requirements as well as network conditions. In contrast to traditional boolean failure detectors (processes are suspected or not), a $\varphi$-failure detector associates a numerical value $\varphi_p$ to every known process $p$, which represents the degree of confidence that process $p$ has crashed.

A preliminary version of this work was presented in [22]. In that version, a stronger partial synchrony model was assumed, namely that of Dwork et al. [10], and only the first algorithm was presented. Assuming the same partial synchrony model, in [23, 25] several ring-based algorithms implementing various classes of failure detectors are proposed, including $\diamond S$ and $\diamond P$. In these algorithms a linear number of bidirectional and unidirectional communication links, respectively, are used forever ($n$ links if no process crashes, which is optimal for $\diamond P$ and unidirectional links). In [24], an algorithm transforming the failure detector class $\diamond C$ into $\diamond P$ is presented. The $\diamond C$ class can be viewed as the combination of classes $\diamond S$ and Omega. The transformation uses $\diamond C$ as a black box, and assumes partially synchronous communication from every process to the leader, and fair communication from the leader to the rest of processes.

## 1.3 Our Contributions

In this paper, we propose three algorithms that implement failure detectors in partially synchronous systems. Two algorithms implement detectors in the class $\diamond S$, while the third one implements a detector in the class $\diamond P$.

As said above, two algorithms implementing $\diamond S$ in a system with weak synchrony are first presented. Both algorithms guarantee that eventually all the correct processes agree permanently on a common correct process, i.e., they implement the Omega failure detector. Then, by not suspecting this common correct process, they obtain the accuracy required by $\diamond S$. Moreover, by suspecting all the other processes, they trivially obtain the completeness required by $\diamond S$. The differences between both $\diamond S$ algorithms are the system requirements to be correct, and the fact that the first one works with up to $n-1$ failures (i.e., it is wait free), while the second one works if up to $f$ processes can crash (and the processes know it). We show that they are both optimal in terms of the number of communication links used forever.

Then, a wait-free algorithm that implements a failure detector of class $\diamond P$ is presented. The algorithm builds on the wait-free $\diamond S$ detector, using the eventually agreed correct process. We show that the algorithm is optimal on the number of bidirectional links used forever.

More specifically, the contributions of this paper are:

- A wait-free algorithm that implements $\diamond S$, by implementing Omega, in a system in which the output links of the correct process with smallest identifier are eventually timely. We show that the maximum number of links that carry messages forever with this algorithm, $n-1$, is in fact optimal.
- An algorithm that implements $\diamond S$, by implementing Omega, in a system in which up

to $f$ processes can fail, with $f \leq n-1$. The algorithm requires that the links connecting the correct process with smallest identifier to the rest among the $f+1$ with smallest identifiers are eventually timely, and the availability of a reliable broadcast service. (A simple way to implement reliable broadcast is by message diffusion, see [7], in a system with reliable communication paths between every pair of correct processes.) The number of links that carry messages forever with this algorithm is at most $f$, which is shown to be optimal.

- A wait free algorithm that implements $\Diamond P$ in a system in which the bidirectional (input and output) links of the correct process with smallest identifier are eventually timely. The number of bidirectional links that carry messages forever is at most $n-1$ with this algorithm, which is shown to be optimal.

It is interesting to compare these results with other results in the literature. For instance, looking at our first algorithm, Aguilera et al. [2, 4] showed that it is possible to implement Omega if *any* process has its output links eventually timely, but at a cost of a quadratic number of links carrying messages forever. To reduce this number of links to $n-1$ the additional assumption of a fair-hub (a node with all links fair) was made. In this paper the additional assumption restricts which is the process whose output links are eventually timely. Similarly, considering our second algorithm, Aguilera et al. [3] implement Omega in a system with fair links (which is known to be equivalent to reliable links) and *some* process whose output links with $f$ processes are eventually timely. Again this comes at the cost of more than $f$ links carrying messages forever. Considering the third algorithm, Larrea et al. [23] and Aguilera et al. [1] have algorithms that implement $\Diamond P$ and have $n$ bidirectional links carrying messages forever. This value is reduced here to $n-1$. Observe that, if (uni)directional links are considered, $\Diamond P$ can be implemented even if only $n$ directional links carry messages forever [25].

## 1.4 Roadmap

The rest of the paper is organized as follows. In Section 2, we describe the system model and discuss different approaches in order to implement failure detectors. In Section 3, we present two optimal algorithms implementing a failure detector of class $\Diamond S$. In Section 4, we present an optimal algorithm implementing a failure detector of class $\diamond P$. Finally, Section 5 concludes the paper.

# 2 The Model

## 2.1 System Model

We consider a distributed system consisting of a finite set $\Pi$ of $n$ processes, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate only by sending and receiving messages. Every pair of processes $(p_i, p_j)$ is assumed to be connected by two directed communication links $(p_i \rightarrow p_j)$ and $(p_j \rightarrow p_i)$, seen also as a bidirectional communication link. We also assume that processes are totally ordered. Without loss of generality, process $p_i$ is preceded by processes $p_1, \ldots, p_{i-1}$, and followed by processes $p_{i+1}, \ldots, p_n$.

Processes can fail by *crashing*, that is, by prematurely halting. Crashes are permanent, i.e., crashed processes do not recover. In every run of the system we identify two complementary subsets of $\Pi$: the subset of processes that do not fail, denoted *correct*, and the subset of processes that do fail, denoted *crashed*. We assume that the number of correct processes in the system in a given run is at least one, i.e., $|correct| \geq 1$.

In the three algorithms presented in this paper, the correct process with smallest identifier is always chosen to have a special role. For that reason we call it the *leader process* and use a special notation for it.

**Definition 1** *$p_{leader}$ is the correct process with smallest identifier, i.e., $leader = \min\{i : p_i \in correct\}$.*

In this regard, we consider asymmetric leader election: always the correct process with the smallest identifier is finally elected. This is not true with other Omega protocols, in which the result depends on the number of suspicions. Moreover, our optimality results are only for this kind of asymmetric protocols.

We use $f$ to denote the maximum number of processes that can crash in any run of the system. If nothing is specified, we assume $f = n-1$. The set of $f+1$ processes with smallest identifiers will be denoted as $P_f$.

We consider a variant of the model of partial synchrony proposed by Chandra and Toueg in [7], which is an adaptation of the models proposed by Dwork et al. in [10]. This model stipulates that, in every run of the system, there is an upper bound on processing delay, defined as the time from the reception of a message to the time the message is processed and (potentially) new messages are sent out. Additionally, some links are *eventually timely,* which means that there is a bound $\delta$ on message transmission times on the links. These bounds are not known *and* they hold only after some unknown but finite time (called *GST* for *Global Stabilization Time*). To simplify the proofs, we will consider that the bound $\delta$ includes both the transmission and processing time of any message sent after *GST*. This can be done without loss of generality due to the upper bound on processing delay. Unless otherwise said, a link that is not eventually timely can be asynchronous and/or lossy.

Each of the three algorithms presented has a different set of timing and reliability requirements from the links of the underlying system. We define them as properties here. First, the wait-free $\Diamond S$ algorithm requires the following property from the system.

**Property 1** *All the output links of* $p_{leader}$ *are eventually timely.*

The $f$-resilient $\Diamond S$ algorithm requires the following property.

**Property 2** *All the output links of* $p_{leader}$ *to the rest of processes in* $P_f$ *are eventually timely, and a reliable broadcast service is available.*

The reliable broadcast service guarantees that a message that has been broadcast will be delivered by all correct processes or none. More precisely, it guarantees that all correct processes deliver the same set of messages. This set includes at least all messages broadcast by correct processes. To provide the reliable broadcast service it is enough to have reliable (or even fair lossy) links. The access to the reliable broadcast service is done with two primitives, R-broadcast($m$) which broadcasts message $m$ in a reliable fashion, and R-deliver($m$) which delivers message $m$. Formally, the reliable broadcast service satisfies the following properties [17]:

- Validity. If a correct process R-broadcasts a message $m$, then it eventually R-delivers $m$.
- Agreement. If a correct process R-delivers a message $m$, then all correct processes eventually R-deliver $m$.

- Uniform integrity. For any message $m$, every process R-delivers $m$ at most once, and only if $m$ was previously R-broadcast by some process.

Finally, the wait-free $\diamond P$ algorithm requires the following property from the system.

**Property 3** *All the bidirectional links of* $p_{leader}$ *are eventually timely.*

All the algorithms presented in this paper assume that a local clock that can accurately measure real-time intervals is available to each process. However, clocks are not synchronized.

## 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 set of processes it suspects to have crashed. These sets can differ from one module to another at a given time. We denote by $suspected_i$ the set 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 set 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 to which they are attached. 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. However, it is assumed that if a process crashes, its failure detector module crashes as well, and vice-versa.

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 algorithms presented in this paper are based on the push model. At any time, at least one process is sending I-AM-ALIVE messages (most of the time we denote them I-AM-THE-LEADER 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 algorithms. We use $\Delta_{i,j}(t)$ to denote the value of $\Delta_{i,j}$ at time $t$.

# 3 Optimal Implementations of $\Diamond S$

## 3.1 Wait-free $\Diamond S$ Algorithm

In this section, we present a first algorithm implementing a failure detector of class $\Diamond S$. The algorithm works in a system in which up to $n-1$ processes can fail (i.e., it is wait free). This algorithm guarantees that eventually all the correct processes converge on the leader process $p_{leader}$ as a common correct process. This property trivially allows the algorithm to provide the eventual weak accuracy property required by $\Diamond S$: eventually, $p_{leader}$ is not suspected by any correct process. The strong completeness property of $\Diamond S$ 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 called $trusted_i$. As we will show, eventually the value of $trusted_i$ for each correct process $p_i$ will be the same, and $trusted_i = leader$.

```
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
cobegin
|| Task 1: repeat periodically
        if $trusted_i = i$ then send I-AM-THE-LEADER to $p_{i+1}, \ldots, p_n$
|| Task 2: when $(trusted_i < i)$ and
                    (did not receive I-AM-THE-LEADER from $p_{trusted_i}$ during the last $\Delta_{i,trusted_i}$ time units)
        $trusted_i \leftarrow trusted_i + 1$
|| Task 3: when (received I-AM-THE-LEADER from $p_j$) and $(j < trusted_i)$
        $trusted_i \leftarrow j$
        $\Delta_{i,j} \leftarrow \Delta_{i,j} + 1$
coend
```

Figure 2: Wait-free algorithm used to implement a failure detector of class $\Diamond S$.

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-THE-LEADER messages periodically (i.e., every $\Delta_{T1}$ time units, with $\Delta_{T1}$ statically defined) to the rest of processes $p_2, \ldots, p_n$. In general, a process $p_i$ will be sending I-AM-THE-LEADER messages periodically to its successors $p_{i+1}, \ldots, p_n$ if $trusted_i = i$ (Task 1). A process $p_i$ such that $trusted_i \neq i$, just waits for periodical I-AM-THE-LEADER messages from the process $p_{trusted_i}$. If it does not receive an I-AM-THE-LEADER message on time (within some timeout period $\Delta_{i,trusted_i}$), then $p_i$ 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-THE-LEADER message from a process $p_j$, such that $j < trusted_i$, then $p_i$ will stop considering 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-THE-LEADER messages periodically, it will automatically stop sending them, since now $trusted_i \neq i$.

### Correctness Proof

We show now that the algorithm of Figure 2, combined with either of the following definitions of $suspected_i$ ($\Pi - \{p_{trusted_i}\}$ or $\Pi - \{p_{trusted_i}, p_i\}$), implements a failure detector of class $\diamond S$. 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 of $\diamond S$, and eventually all crashed processes are permanently suspected by all correct processes, which provides the strong completeness property of $\diamond S$.

Recall that it is assumed that Property 1 holds. 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* on eventually timely links have already been delivered and processed, or lost. These assumptions allow us to consider in the rest of the section that the unknown bounds on processing delay and on message transmission times hold (the later only for the messages sent by $p_{leader}$). 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, namely $p_1, \ldots, p_{leader-1}$, 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) or lost. Then, if at any time $t'' \geq t'$, $trusted_i(t'') = j < leader$, at most $\Delta_{i,j}(t'')$ time units later Task 2 will be activated and the variable $trusted_i$ will be updated to $j + 1$. Hence, there is some time $t_i \geq t'$ at which $trusted_i(t_i) \geq leader$. Since $p_i$ will never receive any other message from processes $p_1, \ldots, p_{leader-1}$ after $t'$, 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:** From the initialization of $trusted_i$ to 1 and Task 2, $\forall t : trusted_{leader}(t) \leq leader$. From Lemma 1, $\forall t > t_0, trusted_{leader}(t) \geq leader$. Hence, $\forall t > t_0, trusted_{leader}(t) = leader$.
☐

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

**Proof:** Follows from Lemma 2 and Task 1. ☐

Let $\Delta_{T1}$ be the period of Task 1. Also, recall that $\delta$ is the maximum time between the sending of a message by $p_{leader}$ and the delivery and processing at its destination process (assuming that the destination is correct).

**Lemma 4** *Let* $p_i \in correct : i \neq leader$*. If at time* $t' > t_0$, $trusted_i(t') > leader$*, then* $\exists t'' : t' < t'' \leq t' + \Delta_{T1} + \delta$ *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-THE-LEADER messages, with a period of $\Delta_{T1}$, to all its successors, including $p_i$. After $t'$, the first I-AM-THE-LEADER message will be sent by $p_{leader}$ at time $t' + \Delta_{T1}$ at the latest. This message takes a maximum time of $\delta$ to be delivered and processed by $p_i$. Hence, at some time $t'' \leq t' + \Delta_{T1} + \delta$, $p_i$ will deliver and process an I-AM-THE-LEADER 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 after $t_0$. 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-THE-LEADER 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-THE-LEADER messages sent by $p_{leader}$ are received and processed by $p_i$ at most $\Delta_{T1} + \delta$ 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 $\Delta_{T1} + \delta$. 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 Lemmas 4 and 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 $\diamond S$.*

### Optimality

In this section, we study the number of communication links used forever by the algorithm. Observe that, eventually, only $p_{leader}$ sends messages. This means that at most its $n-1$ output links carry messages forever.

We prove now that $n-1$ is in fact a lower bound on the number of unidirectional links that carry messages forever in any fault-free execution of a $\Diamond S$ algorithm, if up to $n-1$ processes can crash. Hence this algorithm is optimal with respect to this parameter.

**Theorem 2** *Let $A$ be any (wait-free) algorithm that implements $\Diamond S$ in a system in which up to $n-1$ processes can crash. Then, in all fault-free runs of $A$, at least $n-1$ unidirectional links carry messages forever.*

**Proof:** Consider some such algorithm $A$ and assume that it has a fault-free run $R$ in which no more than $n-2$ unidirectional links carry messages after some time $T$. Then, after $T$, the set of processes can be divided into at least two disjoint non-empty subsets of processes such that each subset $\Pi_k$ does not communicate anymore with the rest of processes $\Pi \setminus \Pi_k$ (they are permanently disconnected).

From the eventual weak accuracy property, there must be a process $p_\ell$ and a time after which $p_\ell \notin suspected_i$ permanently, for each process $p_i$. Let $\Pi_j$ be the subset that contains $p_\ell$. Consider a run $R'$ in which every process behaves exactly like in $R$ except that the whole set $\Pi_j$ crashes simultaneously after time $T$. Since $\Pi_j$ is disconnected from the rest of processes, no process in $\Pi \setminus \Pi_j$ notices the failures. Then, since $p_\ell$ is never permanently suspected by the processes in $\Pi \setminus \Pi_j$, strong completeness is not satisfied. $\Box$

**Corollary 2** *The algorithm of Figure 2 is optimal on the number of unidirectional links that carry messages forever among the algorithms that implement $\Diamond S$ in systems with up to $n-1$ crashes.*

## 3.2 $f$-Resilient $\Diamond S$ Algorithm

In this section, we present a second algorithm that implements a $\Diamond S$ failure detector. The main differences of this algorithm with respect to the previous is that it uses the knowledge of the maximum number $f$ of processes that can crash to increase the efficiency, and that it has different requirements from the system.

In fact, this algorithm uses the same approach as the previous one to choose a leader, but instead of running it on the whole set of processes, it only uses $f+1$ processes (in particular, $P_f = \{p_1, ..., p_{f+1}\}$). This guarantees the existence of at least one correct process (at least $p_{leader}$) in such a set. Then, every time a leader is chosen, it is communicated to the rest of processes, which adopt it as their trusted process. Figure 3 presents the algorithm in detail.

```
Every process $p_i, i = 1, \ldots, f+1$ executes:
$trusted_i \leftarrow 1$; $count_i \leftarrow 0$
$\forall j \in \{1, \ldots, i-1\} : \Delta_{i,j} \leftarrow$ default timeout
cobegin
|| Task 1: repeat periodically
    if $trusted_i = i$ then send I-AM-THE-LEADER to $p_{i+1}, \ldots, p_{f+1}$
|| Task 2: when ($trusted_i < i$) and
                (did not receive I-AM-THE-LEADER from $p_{trusted_i}$ during the last $\Delta_{i,trusted_i}$ time units)
    $trusted_i \leftarrow trusted_i + 1$
    if $trusted_i = i$ then R-broadcast(NEW-LEADER, $count_i$)
|| Task 3: when (received I-AM-THE-LEADER from $p_j$) and ($j < trusted_i$)
    $\Delta_{i,j} \leftarrow \Delta_{i,j} + 1$
    $trusted_i \leftarrow j$
|| Task 4: when (R-deliver(NEW-LEADER, $count_j$) from $p_j$) and (($count_i, i$) < ($count_j, j$))
    $count_i \leftarrow count_j + 1$
    if $trusted_i = i$ then R-broadcast(NEW-LEADER, $count_i$)
coend
Every process $p_i, i = f+2, \ldots, n$ executes:
$trusted_i \leftarrow 1$
$count_i \leftarrow 0$
cobegin
|| Task 1: when (R-deliver(NEW-LEADER, $count_j$) from $p_j$) and (($count_i, trusted_i$) < ($count_j, j$))
    $trusted_i \leftarrow j$
    $count_i \leftarrow count_j$
coend
```

Figure 3: $f$-resilient algorithm used to implement a failure detector of class $\Diamond S$.

The difficulty here is to make sure that the communication of leaders to the rest of processes is done in such a way that correctness is guaranteed. For that, we use the reliable broadcast service that is available by Property 2. We use reliable broadcast to enforce that the last message R-delivered by the processes not in $P_f$ was sent by $p_{leader}$, the process trusted by all processes in $P_f$. To do so, processes that believe to be the leader R-broadcast a NEW-LEADER message to the rest of processes to announce so. We impose an order among these messages by making them to carry a counter (i.e., a scalar clock for leader proposals), and breaking ties with the sender's identifier. (In Figure 3 we assume $(count_i, i) < (count_j, j)$ if $count_i < count_j$ or both $count_i = count_j$ and $i < j$.) Processes send such a message when they become potential leaders (Task 2) or when they are leaders and find that another process sent a message with larger counter (Task 4). The processes that are not in $P_f$ apply these messages in increasing order. To prove the correctness of the algorithm we just show that the last such message, i.e., the message with the highest associated counter, was sent by $p_{leader}$.

### Correctness Proof

We show now that the algorithm of Figure 3 implements a failure detector of class $\Diamond S$. Observe that $p_{leader}$ is always in the set $P_f$. The algorithm of Figure 3 uses the same approach to choose a leader as the algorithm of Figure 2, but just among the processes in $P_f$ instead of the whole set of processes $\Pi$. Additionally, Property 2 provides an assumption for $P_f$ similar to the one that Property 1 provides for $\Pi$. Hence, applying a similar reasoning to that of the algorithm of Figure 2, it is simple to prove that, eventually, all the correct processes in $P_f$ will permanently agree on the same leader, and that this leader will be $p_{leader}$. Hence, the following lemma holds.

**Lemma 6** $\exists t : \forall t' > t, \forall p_i \in correct \cap P_f, trusted_i(t') = leader.$

In order to prove that the rest of correct processes will also agree permanently on $p_{leader}$, we will show that the NEW-LEADER message with the largest counter R-delivered to all correct processes to announce a leader —if any NEW-LEADER message is R-delivered— was sent by $p_{leader}$.

**Lemma 7** *If any* NEW-LEADER *message is R-delivered by the correct processes, then the R-delivered* NEW-LEADER *message with the largest counter was R-broadcast by process* $p_{leader}$.

**Proof:** Assume by contradiction that the R-delivered NEW-LEADER message with the largest counter was R-broadcast by a process $p_j$ with $j \neq leader$. From the properties of reliable broadcast, $p_{leader}$ will R-deliver the message. When it does so, it sets $count_{leader}$ to $count_j + 1$.

There are two cases to consider. If $p_{leader}$ had $trusted_{leader} = leader$ when it R-delivered the message, then it R-broadcasts a new (NEW-LEADER, $count_j + 1$) message. If that was not the case, from Lemma 6, eventually $p_{leader}$ sets $trusted_{leader} = leader$, and R-broadcasts a new (NEW-LEADER, $count$) message, with $count > count_j$. In either case, the corresponding message gets R-delivered, which contradicts the initial assumption. □

**Lemma 8** *$\exists t : \forall t' > t, \forall p_i \in correct, f + 2 \leq i \leq n, trusted_i(t') = leader$.*

**Proof:** Follows directly from Lemma 7, if some NEW-LEADER message is ever R-delivered. If no NEW-LEADER message is R-delivered, it is because $leader = 1$, and the claim follows from the way processes $p_{f+2}, \ldots, p_n$ (initially) set their trusted process in the algorithm. □

**Theorem 3** *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 3, combined with either of these definitions of $suspected_i$, implements a failure detector of class $\diamond S$.*

**Proof:** Follows directly from Lemma 6 and Lemma 8. □

### Optimality

Observe in the algorithm of Figure 3 that, once the last (if any) NEW-LEADER message is R-delivered, all the messages sent are from $p_{leader}$ to the rest of processes in $P_f$. Then, at most $f$ links carry messages forever. We prove now that $f$ is in fact a lower bound on the number of links that carry messages forever in any fault-free execution of a $\diamond S$ algorithm if up to $f$ processes can crash. Hence this algorithm is optimal with respect to this parameter.

**Theorem 4** *Let $A$ be any algorithm that implements $\diamond S$ in a system in which up to $f$ processes can crash. Then in all fault-free runs of $A$ at least $f$ links carry messages forever.*

**Proof:** Consider some such algorithm $A$ and assume that it has a fault-free run $R$ in which no more than $f - 1$ links carry messages after some time $T$. Then, after $T$, the set of processes can be divided into at least $n - f + 1$ disjoint non-empty subsets of processes such that each subset $\Pi_k$ does not communicate anymore with the rest of processes $\Pi \setminus \Pi_k$ (they are permanently disconnected). Observe that no subset has size larger than $n-(n-f)=f$.

From the eventual weak accuracy property, there must be a process $p_\ell$ and a time after which $p_\ell \notin suspected_i$ permanently, for each process $p_i$. Let $\Pi_j$ be the subset that contains $p_\ell$. Consider a run $R'$ in which every process behaves exactly like in $R$ except that the whole set $\Pi_j$ crashes simultaneously after time $T$. Since $\Pi_j$ is disconnected from the rest of processes, no process in $\Pi \setminus \Pi_j$ notices the failures. Then, since $p_\ell$ is never permanently suspected by the processes in $\Pi \setminus \Pi_j$, strong completeness is not satisfied. $\square$

**Corollary 3** *The algorithm of Figure 3 is optimal on the number of links that carry messages forever among the algorithms that implement $\diamond S$ in systems with up to $f$ crashes.*

# 4 Wait-free $\diamond P$ Algorithm

In this section, we propose an algorithm implementing a failure detector of the Eventually Perfect class ($\diamond P$). This algorithm successfully exploits the eventual leader election property of the wait-free $\diamond S$ algorithm of the previous section, and extends it with a periodic communication between every non-leader process and its leader process. As there is eventually a unique and correct leader, it can be used to build and propagate a global set of suspected processes satisfying the properties of $\diamond P$.

Figure 4 presents the algorithm in detail, which works as follows. Each *leader* process (i.e., each process that trusts itself) builds a local set of suspected processes by using timeouts (Tasks 2, 4 and 5), and sends its set periodically to the rest of processes (Task 1). Concurrently, each non-leader process periodically sends an I-AM-ALIVE message to its trusted process (Task 1). Finally, when a process receives a set of suspected processes from its trusted process, it adopts this set as its own set (Task 3).

While the algorithms implementing $\diamond S$ of the previous section require that, eventually, the bound on message transmission times holds only for the output links of the leader process to the rest of correct processes (in the case of the second algorithm, to the rest of correct processes in $P_f$), this algorithm requires that the bound holds also for the links $(p_i \rightarrow p_{leader})$, for every correct process $p_i$ (except $p_{leader}$). Not surprisingly, the fact that

```
Every process $p_i, i = 1, \dots, n$ executes:
$trusted_i \leftarrow 1$
$suspected_i \leftarrow \emptyset$                    $\{suspected_i$ provides the properties of $\Diamond P\}$
$\forall j \in \{1, \dots, n\} : \Delta_{i,j} \leftarrow$ default timeout$\{\Delta_{i,j}, j < i$ are used to eventually agree on a common leader process$\}$
                    $\{\Delta_{i,j}, j > i$ are used by the leader to build the set of suspected processes$\}$
cobegin
|| Task 1: repeat periodically
    if $trusted_i = i$ then
        send (I-AM-THE-LEADER, $suspected_i$) to $p_{i+1}, \dots, p_n$
    else
        send I-AM-ALIVE to $p_{trusted_i}$
|| Task 2: when ($trusted_i < i$) and (did not receive (I-AM-THE-LEADER, $suspected_{trusted_i}$) from $p_{trusted_i}$
                    during the last $\Delta_{i,trusted_i}$ time units)
    $trusted_i \leftarrow trusted_i + 1$
    if $trusted_i = i$ then $suspected_i \leftarrow \{p_1, \dots, p_{i-1}\}$
|| Task 3: when (received (I-AM-THE-LEADER, $suspected_j$) from $p_j$) and ($j \leq trusted_i$)
    if $j < trusted_i$ then
        $trusted_i \leftarrow j$
        $\Delta_{i,j} \leftarrow \Delta_{i,j} + 1$
    $suspected_i \leftarrow suspected_j$
|| Task 4: when ($trusted_i = i$) and (did not receive I-AM-ALIVE from $p_j$ during the last $\Delta_{i,j}$ time units)
                    and ($j > i$)
    $suspected_i \leftarrow suspected_i \cup \{p_j\}$
|| Task 5: when ($trusted_i = i$) and (received I-AM-ALIVE from $p_j$) and ($p_j \in suspected_i$)
    $suspected_i \leftarrow suspected_i - \{p_j\}$
    $\Delta_{i,j} \leftarrow \Delta_{i,j} + 1$
coend
```

Figure 4: Wait-free algorithm implementing $\Diamond P$.

the class $\Diamond P$ of failure detectors is strictly stronger than $\Diamond S$ is reflected in this stronger synchrony requirement.

### Correctness Proof

We show now that the algorithm of Figure 4 implements a failure detector of class $\Diamond P$. Note first that, concerning the management of the $trusted_i$ variable, the first three tasks of the algorithm of Figure 4 are equivalent to the three tasks of the algorithm of Figure 2. Note also that the rest of the algorithm of Figure 4 does not affect the $trusted_i$ variable. Hence the following observation.

**Observation 1** *Theorem 1 holds with the algorithm of Figure 4.*

This theorem states that eventually all the correct processes will permanently trust the same correct process $p_{leader}$. All the time instants considered in the rest of the proof are greater than $t_1$, as defined in Theorem 1. We also assume that all the incorrect processes have already crashed, and all their messages have already been delivered and processed or lost.

**Lemma 9** *Eventually every process that crashes is permanently suspected by* $p_{leader}$.

**Proof:** Let $p_j$ be a process that crashes. There are two cases to consider: (1) $j < leader$, and (2) $j > leader$. In case (1), by Task 2 $p_{leader}$ will include $p_j$ in its set of suspected processes as soon as it trusts itself. In case (2), due to its crash, $p_j$ will stop sending I-AM-ALIVE messages. From Task 4, $p_{leader}$ will eventually include $p_j$ in its set of suspected processes. For $p_j$ to be removed from that set of suspected processes, $p_{leader}$ has to receive an I-AM-ALIVE message from $p_j$. Since $p_j$ has crashed, this will not happen, and thus $p_{leader}$ will permanently suspect $p_j$. □

Let $\Delta_{T1}$ be the period of Task 1. Also, since Property 3 holds, recall that $\delta$ is the maximum time between the sending of a message by $p_{leader}$ and the delivery and processing by its destination process (assuming that the destination is correct), as well as the maximum time between the sending of a message by the rest of correct processes to $p_{leader}$ and the delivery and processing by $p_{leader}$.

**Lemma 10** *Let* $p_i \in correct, i \neq leader$*:* $p_{leader}$ *will suspect process* $p_i$ *a finite number of times.*

**Proof:** Let us assume, by the way of contradiction, that $p_{leader}$ suspects $p_i$ an infinite number of times. From Task 1 and Theorem 1, $p_i$ will eventually and permanently send I-AM-ALIVE messages periodically to $p_{leader}$. From Task 5, each time that $p_i$ is incorrectly suspected by $p_{leader}$ (in Task 4), $p_{leader}$ will eventually stop suspecting $p_i$, incrementing its timeout value $\Delta_{leader,i}$. Hence, since this will happen an infinite number of times, eventually $\Delta_{leader,i}$ will be larger than $(\Delta_{T1} + \delta)$. However, after that happens $p_{leader}$ will no more suspect $p_i$, which is a contradiction. □

**Lemma 11** *There is a time after which no correct process is suspected by* $p_{leader}$.

**Proof:** Since a process never suspects itself, the lemma directly applies to the process $p_{leader}$ itself. For the rest of correct processes, it follows directly from Lemma 10 and the fact that by the algorithm (Tasks 1, 4, and 5) all the incorrect suspicions made by $p_{leader}$ are eventually corrected. □

**Lemma 12** *Eventually every correct process will permanently agree with $p_{leader}$ in the set of suspected processes.*

**Proof:** From Task 1, $p_{leader}$ will send periodically its set of suspected processes to every correct process. Let $p_j$ be a correct process. From Task 3 and Theorem 1, $p_j$ will receive periodically the set of suspected processes of $p_{leader}$, adopting it as its own set of suspected processes. □

**Theorem 5** *The algorithm of Figure 4 implements a failure detector of class $\diamond P$.*

**Proof:** From Lemmas 9, 11, and 12, 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). This gives us the two properties of $\diamond P$. □

### Optimality

The algorithm of Figure 4 has at most $n-1$ bidirectional links that carry messages forever, i.e., the input and output links of the leader process. We prove now that, if only bidirectional links are available, $n-1$ is in fact a lower bound on the number of bidirectional links that carry messages forever in any fault-free execution of a $\diamond P$ algorithm if up to $n-1$ processes can crash. Hence this algorithm is optimal with respect to this parameter. Note however that the optimality is only about bidirectional links. If we count each bidirectional link as two unidirectional links, the algorithm is not optimal.

**Theorem 6** *Let $A$ be any algorithm that implements $\diamond P$ in a system in which up to $n-1$ processes can crash. Assume that only bidirectional links are available. Then, in all fault-free runs of $A$ at least $n-1$ bidirectional links carry messages forever.*

**Proof:** The proof is almost verbatim to that of Theorem 2. $\square$

**Corollary 4** *The algorithm of Figure 4 is optimal on the number of bidirectional links that carry messages forever among the algorithms that implement $\diamond P$ in systems with up to $n-1$ crashes.*

## 5 Conclusion

In this paper, we have presented two algorithms implementing $\diamond S$, the weakest failure detector class for solving Consensus. We have also presented an algorithm implementing a failure detector of class $\diamond P$. Our algorithms are optimal in terms of the number of communication links used forever.

Comparing to other algorithms that implement $\diamond S$, it may seem that our $\diamond S$ algorithms have a big loss of accuracy, because all processes except one are systematically suspected. However, the fact that eventually all the processes agree on a leader process can be very helpful to solve Consensus more efficiently, i.e., in less rounds, than existing previous algorithms for $\diamond S$ [24, 32].

## References

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC'2001)*, pages 108–122, Lisbon, Portugal, October 2001. LNCS 2180, Springer-Verlag.

[2] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing $\Omega$ with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'2003)*, pages 306–314, Boston, Massachusetts, July 2003.

[3] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd*

*ACM Symposium on Principles of Distributed Computing (PODC'2004)*, St. John's, Newfoundland, Canada, July 2004.

[4] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, October 2008.

[5] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'2002)*, pages 354–363, Washington D.C., June 2002.

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

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

[8] W. Chen, S. Toueg, and M. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[9] F. Chu. Reducing $\Omega$ to $\diamond\mathcal{W}$. *Information Processing Letters*, 67(6):289–293, September 1998.

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

[11] A. Fernández, E. Jiménez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *DSN*, pages 166–178. IEEE Computer Society, 2006.

[12] A. Fernández Anta and M. Raynal. From an intermittent rotating star to a leader. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *OPODIS*, volume 4878 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2007.

[13] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proceedings of the 8th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'2001)*, pages 146–153, Seoul, Korea, December 2001.

[14] 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.

[15] 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, Bad Neuenahr, Germany, September 1995.

[16] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'2001)*, pages 170–179, Newport, Rhode Island, August 2001.

[17] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Addison-Wesley, 2nd edition, 1993.

[18] N. Hayashibara, X. Défago, and T. Katayama. Two-ways adaptive failure detection with the $\varphi$-failure detector. In *Proceedings of the Workshop on Adaptive Distributed Systems (WADIS)*, pages 22–27, Sorrento, Italy, October 2003.

[19] 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.

[20] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Chasing the weakest system model for implementing $\Omega$ and consensus. *IEEE Transactions on Dependable and Secure Computing*, 2008. To appear.

[21] E. Jiménez, S. Arévalo, and A. Fernández. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60–63, 2006.

[22] M. Larrea, A. Fernández, and 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)*, pages 52–59, Nurenberg, Germany, October 2000.

[23] M. Larrea, A. Fernández, and S. Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 53(7):815–828, 2004.

[24] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. *J. Parallel Distrib. Comput.*, 65(3):361–373, 2005.

[25] M. Larrea, A. Lafuente, I. Soraluze, R. Cortiñas, and J. Wieland. On the implementation of communication-optimal failure detectors. In *Proceedings of the Third Latin-American Symposium on Dependable Computing (LADC'2007)*, pages 25–37, Morelia, Mexico, September 2007. LNCS 4746, Springer-Verlag.

[26] D. Malkhi, F. Oprea, and L. Zhou. *Omega* meets paxos: Leader election and stability without eventual timely links. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2005.

[27] A. Mostéfaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'2003)*, pages 351–360, San Francisco, California, June 2003.

[28] A. Mostéfaoui, E. Mourgaya, M. Raynal, and C. Travers. A time-free assumption to implement eventual leadership. *Parallel Processing Letters*, 16(2):189–208, 2006.

[29] A. Mostéfaoui, D. Powell, and M. Raynal. A hybrid approach for building eventually accurate failure detectors. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'2004)*, pages 57–65, Papeete, Tahiti, March 2004.

[30] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and C. Travers. From omega to Omega: A simple bounded quiescent reliable broadcast-based transformation. *Journal of Parallel and Distributed Computing*, 67(1):125–129, January 2007.

[31] A. Mostéfaoui 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, Bratislava, Slovak Republic, September 1999. LNCS 1693, Springer-Verlag.

[32] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[33] A. Mostéfaoui, M. Raynal, and C. Travers. Crash-resilient time-free eventual leadership. In *Proceedings of the 23rd International IEEE Symposium on Reliable Distributed Systems (SRDS'2004)*, pages 208–217, Florianpolis, Brazil, October 2004.

[34] A. Mostéfaoui, M. Raynal, and C. Travers. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):656–666, 2006.

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

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