Crash-resilient Time-free Eventual Leadership

Achour MOSTEFAOUI Michel RAYNAL Corentin TRAVERS

IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
{achour|raynal|travers}@irisa.fr

Abstract

Leader-based protocols rest on a primitive able to provide the processes with the same unique leader. Such protocols are very common in distributed computing to solve synchronization or coordination problems. Unfortunately, providing such a primitive is far from being trivial in asynchronous distributed systems prone to process crashes. (It is even impossible in fault-prone purely asynchronous systems.) To circumvent this difficulty, several protocols have been proposed that build a leader facility on top of an asynchronous distributed system enriched with synchrony assumptions. This paper consider another approach to build a leader facility, namely, it considers a behavioral property on the flow of messages that are exchanged. This property has the noteworthy feature not to involve timing assumptions. Two protocols based on this time-free property that implement a leader primitive are described. The first one uses potentially unbounded counters, while the second one (which is a little more involved) requires only finite memory. These protocols rely on simple design principles that make them attractive, easy to understand and provably correct.

Keywords*: Asynchronous system, Distributed algorithm, Fault tolerance, Leader, Process crash, Time-free Protocol.*

1 Introduction

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

While consensus is considered as a "theoretical" problem, middleware designers are usually interested in the more practical *Atomic Broadcast* problem. That problem is both a communication problem and an agreement problem. Its communication part specifies that the processes can broadcast and deliver messages in such a way that each correct[1] process delivers at least the messages sent by the correct processes. Its agreement part specifies that there is a single delivery order (so, the correct processes deliver the same sequence of messages, and a faulty process delivers a prefix of this sequence of messages). It has been shown that *consensus* and *atomic broadcast* are equivalent problems in asynchronous systems prone to process crashes [4]: in such a setting, any protocol solving one of them can be used as a black box on top of which the other problem can be solved. Consequently, in asynchronous distributed systems prone to process crashes, the impossibility of solving consensus extends to atomic broadcast.

When faced to process crashes in an asynchronous distributed system, the main problem comes from the fact that it is impossible to safely distinguish a crashed process from a process that is slow or with which communication is very slow. To overcome this major difficulty, Chandra and Toueg have introduced the concept of *Unreliable Failure Detector* [4]. A failure detector can be seen as an oracle [15] made up of a set of modules, each associated with a process. The failure detector module attached to a process provides it with a list of processes it suspects of having crashed. A failure detector module can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. In their seminal paper [4], Chandra and Toueg have introduced several classes of failure detectors. A class is defined by two abstract properties, namely a *Completeness* property and an *Accuracy* property. Completeness is on the actual detection of crashes, while accuracy restricts erroneous suspicions. As an example, the class of failure detectors denoted $\diamond\mathcal{S}$ includes all failure detectors such that (1) eventually each crashed process is permanently suspected by every correct process, and (2) there is a correct process that, after some finite but unknown time, is never suspected by the correct processes (accuracy). Interestingly, several protocols that solve the consensus problem in asynchronous distributed systems augmented with a failure

[1] A *correct* process is a process that does not crash. See Section 2.

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

detector of the class $\diamond\mathcal{S}$, and including a majority of correct processes, have been designed (e.g., [4, 11, 12, 18] to cite a few). It has been shown that $\diamond\mathcal{S}$ is the weakest class of failure detectors that allow solving the consensus problem in an asynchronous system prone to process crashes (with the additional assumption that a majority of processes are correct) [5].

A facility that is at the core of several distributed agreement protocols is the class of leader oracles (usually denoted Ω). Such an oracle offers a leader() primitive that satisfies the following leadership property: a unique correct leader is eventually elected, but there is no knowledge on when this common leader is elected and, before this occurs, several distinct leaders (possibly conflicting) can co-exist. Interestingly, it is possible to solve consensus (and related agreement problems) in asynchronous distributed system equipped with such a "weak" oracle (as soon as these systems have a majority of correct processes) [5, 20]. It has also been shown that, as far as failure detection is concerned, Ω and $\diamond\mathcal{S}$ have the same computational power in asynchronous distributed system prone to process crashes [5, 6].

Neither $\diamond\mathcal{S}$ nor Ω can be implemented in pure (time-free) asynchronous systems (their implementation would contradict the consensus impossibility result [9]). Nevertheless, these oracle classes allow the protocols that use them to benefit from a very nice property, namely *indulgence* [10]. Let P be an oracle-based protocol, and PS be the safety property satisfied by its outputs. P is *indulgent with respect to its underlying oracle* if, whatever the behavior of the oracle, its outputs never violate the safety property PS. This means that each time P produces outputs, those are correct. Moreover, P always produces outputs when the underlying oracle meets its specification. The only case where P can be prevented from producing outputs is when the underlying oracle does not meet its specification. (Let us notice that it is still possible that P produces outputs despite the fact that its underlying oracle does not work correctly.)

Interestingly, $\diamond\mathcal{S}$ and Ω are classes of oracles that allow the design of indulgent consensus protocols [11]. It is important to notice that indulgence is a first class property that makes valuable the design of "approximate" protocols that do their best to implement $\diamond\mathcal{S}$ or Ω on top of the asynchronous system itself. The periods during which their best effort succeeds in producing a correct implementation of the oracle are called "good" periods, the upper layer oracle-based protocol P then produces outputs and those are correct. During the other periods (sometimes called "bad" periods), P does not produce erroneous outputs. The only bad thing that can happen in a bad period is that P can be prevented from producing outputs. It is important to notice that neither the occurrence, nor the length of the good/bad periods (sometimes called stable vs unstable periods) can be known by the upper layer protocol P that uses the underlying oracle. The only thing that is known is that a result produced by P is always correct.

The fact that the safety property PS of the $\diamond\mathcal{S}$/Ω-based protocol P can never be violated, and the fact that its liveness property (outputs are produced) can be ensured in "good" periods, make attractive the design of indulgent $\diamond\mathcal{S}$/Ω-based protocols, and motivates the design of underlying "best effort" protocols that implement a $\diamond\mathcal{S}$ or Ω oracle within the asynchronous distributed system itself. A challenge is then to identify properties that, when satisfied by the asynchronous system, ensure that it evolves in a good period.

Related work Several works have considered the implementation of failure detectors of the class $\diamond\mathcal{S}$ or Ω (e.g., [1, 4, 8, 14, 22]). Basically, all these works consider that, eventually, the underlying system (or a part of it) behaves in a synchronous way. More precisely, some of these implementations consider the *partially synchronous system* model [4] which is a generalization of the models proposed in [7]. A partially synchronous system assumes there are bounds on process speeds and message transfer delays, but these bounds are not known and hold only after some finite but unknown time (called *Global Stabilization Time*). The protocols implementing failure detectors in such systems obey the following principle: using successive approximations, each process dynamically determines a value Δ that eventually becomes an upper bound on transfer delays and processing speed.

The Ω protocol described in [1] considers weaker synchrony assumptions, namely it requires synchronous processes (process speed is bounded) and the existence of at least one correct process whose output links are eventually timely (i.e., there are a bound δ and a time t, such that, after t, each message sent on such a link is received within δ time). The Ω protocol described in [2] improves on the previous one as it requires that only f output links of a correct process be eventually timely (where f is the upper bound on the number of faulty processes).

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

Assuming that each process can broadcast queries and then, for each query, wait for the corresponding responses, we say that a response to a query is a *winning* response if it arrives among the first $(n - f)$ responses to that query (the other responses to that query are called *losing* responses). Let us consider the following behavioral property: "There

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

are a correct process p_i and a set Q of $(f+1)$ processes such that eventually the response of p_i to each query issued by any $p_j \in Q$ is always a winning response (until -possibly- the crash of p_j)". It is shown in [16] that failure detectors of the class $\diamond\mathcal{S}$ can be implemented when this property is satisfied. This means that it is possible to design a protocol satisfying the completeness and accuracy properties of $\diamond\mathcal{S}$ on top of asynchronous distributed systems satisfying the previous requirement. Interestingly, such a requirement does not involve bounds on communication times (they can be arbitrary). A probabilistic analysis for the case $f = 1$ shows that such a behavioral property on the message exchange pattern is practically always satisfied [16].

Let MP be the previous behavioral property on the message exchange pattern. This paper investigates MP and shows how it can be used to implement a leader oracle. It is important to notice that the MP property is time-free: it does not involve timing assumptions. It that sense, the protocols presented in this paper show that, as soon as the MP property is satisfied by the message exchange pattern, the eventual leader election problem can be solved in asynchronous systems prone to process crashes without requiring dependable timeout values. The paper presents two protocols. The first uses unbounded counters. The second improves it in the sense that it uses only finite memory. So, the eventual leader protocol we finally obtain is a time-free finite memory protocol.

Organization of the paper The paper is made up of six sections. Section 2 defines the system model and the behavioral assumption MP. Section 3 defines the class of eventual leader oracles, and shows that they cannot be implemented in purely asynchronous system. Then, Sections 4 and 5 present MP-based protocols that implement an eventual leader facility (the first uses unbounded counters, while the second needs only finite memory). Finally, Section 6 concludes the paper.

2 System Model and Additional Assumption

2.1 System Model

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

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

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

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

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

In the following $AS_{n,f}[\emptyset]$ denotes an asynchronous distributed system made up of n processes among which up to $f < n$ can crash ($1 \leq f < n$).

2.2 A Behavioral Property on the Message Exchange Pattern

As implementing a leader oracle in an asynchronous system is impossible (see Theorem 1), we consider the following additional assumption that we call MP:

> "There are a time t, a correct process p_i and a set Q of $(f + 1)$ processes (t, p_i and Q are not known in advance) such that, after t, each process $p_j \in Q$ gets a winning response from p_i to each of its queries (until p_j possibly crashes)."

The intuition that underlies this property is the following. Even if the system never behaves synchronously during a long enough period, it is possible that its behavior has some "regularity" that can be exploited to build a leader oracle. This regularity can be seen as some "logical synchrony" (as

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

opposed to "physical" synchrony). More precisely, MP states that, eventually, there is a cluster Q of $(f+1)$ processes that (until some of them possibly crash) receive winning responses from p_i to their queries. This can be interpreted as follows: among the n processes, there is a process that has $(f+1)$ "favorite neighbors" with which it communicates faster than with the other processes. When we consider the particular case $f = 1$, MP boils down to a simple channel property, namely, there is channel (p_i, p_j) that is never the slowest among the channels connecting p_j to the other processes (it is shown in [16] that the probability that this property be satisfied in practice is very close to 1).

In the following, $AS_{n,f}[MP]$ denotes an asynchronous distributed system made up of n processes among which up to f can crash ($1 \leq f < n$), and satisfying the property MP.

3 A Leadership Facility

Definition and Use A *leader* oracle is a distributed entity that provides the processes with a function leader() that returns a process name each time it is invoked. A unique correct leader is eventually elected but there is no knowledge of when the leader is elected. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this "anarchy" period is over. The *leader* oracle (denoted Ω) satisfies the following property[2]:

- Eventual Leadership: There is a time t and a correct process p such that, after t, every invocation of leader() by any correct process returns p.

Ω-based consensus algorithms are described in [11, 13, 20][3] for systems where a majority of processes are correct ($f < n/2$). Such consensus algorithms can then be used as a subroutine to implement atomic broadcast protocols (e.g., [4, 13, 19]).

An Impossibility Result As consensus can be solved in an asynchronous system with a majority of correct processes, and equipped with a leader oracle, and as consensus cannot be solved in purely asynchronous systems [9], it follows that a leader oracle cannot be implemented in an asynchronous system $AS_{n,f}[\emptyset]$ with $1 \leq f < n/2$. The theorem that follows shows a more general result in the sense that it does not state a constraint on f.

Theorem 1 *No leader oracle can be implemented in* $AS_{n,f}[\emptyset]$ *with* $1 \leq f < n$.

[2]This property refers to a notion of global time. This notion is not accessible to the processes.

[3]The Paxos protocol [13] is leader-based and considers a more general model where processes can crash and recover, and links are fair lossy. (Its first version dates back to 1989, i.e., before the Ω formalism was introduced.)

Proof [4] The proof is by contradiction. Assuming that there is a protocol implementing a leader oracle, we construct a crash-free execution in which there is an infinite sequence of leaders such that any two consecutive leaders are different, from which it follows that the eventual leadership property is not satisfied.

- Let R_1 be a crash-free execution, and t_1 be the time after which some process p_{ℓ_1} is elected as the definitive leader.

  Moreover, let R'_1 be an execution identical to R_1 until $t_1 + 1$, and where p_{ℓ_1} crashes at $t_1 + 2$.

- Let R_2 be a crash-free execution identical to R'_1 until $t_1 + 1$, and where the messages sent by p_{ℓ_1} after $t_1 + 1$ are arbitrarily delayed (until some time that we will specify later).

  As, for any process $p_x \neq p_{\ell_1}$, R_2 cannot be distinguished from R'_1, it follows that some process $p_{\ell_2} \neq p_{\ell_1}$ is elected as the definitive leader at some time $t_2 > t_1$. After p_{ℓ_2} is elected, the messages from p_{ℓ_1} can be received.

  Moreover, let R'_2 be an execution identical to R_2 until $t_2 + 1$, and where p_{ℓ_2} crashes at $t_2 + 2$.

- Let R_3 be a crash-free execution identical to R'_2 until $t_2 + 1$, and where the messages from ℓ_2 are delayed (until some time that we will specify later).

  Some process $p_{\ell_3} \neq p_{\ell_2}$ is elected as the definitive leader at some time $t_3 > t_2 > t_1$. After p_{ℓ_3} is elected, the messages from p_{ℓ_2} are received. Etc.

This inductive process, repeated indefinitely, constructs a crash-free execution in which an infinity of leaders are elected at times $t_1 < t_2 < t_3 < \ldots$ and such that no two consecutive leaders are the same process. Hence, the eventual leadership property we have assumed is not satisfied. $\square_{Theorem\ 1}$

4 An MP-based Leader Protocol

4.1 Underlying Principles

The protocol is made up of three tasks executed by each process. Its underlying principles are relatively simple. It is based on the following heuristic: each process elects as a leader the process it suspects the least. To implement this idea, each process p_i manages an array $count_i[1..n]$ in such a way that $count_i[j]$ counts the number of times p_i suspects

[4]This proof is close to the proof given in [3] where we show that there is no protocol implementing a failure detector of the class $\diamond\mathcal{S}$ in $AS_{n,f}[\emptyset]$ with $1 \leq f < n$.

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

p_j to have crashed. Then, if $count_i[j]$ never stops increasing, p_i heuristically considers that p_j has crashed. According to this management of its $count_i$ array, p_i considers that its current leader is the process p_ℓ such that $count_i[\ell]$ has the smallest value[5] (see Task $T3$).

The aim of the task $T1$ and $T2$ is to manage the array $count_i$ such that the previous heuristic used to define the current leader be consistent, i.e., satisfies the eventual leadership property. To benefit from the MP property, the task $T1$ uses the underlying query-response mechanism. Periodically, each p_i issues a query and waits for the $(n - f)$ corresponding winning responses (lines 101-102). The response from p_j carries the set of processes that sent winning responses to its last query (this set is denoted rec_from_j). Then, according to the rec_from_j sets it has received, p_i updates accordingly its $count_i$ array.

The QUERY_ALIVE() messages implementing the query-response mechanism are used as a gossiping mechanism to disseminate the value of the $count_i$ array of each process p_i. The aim of this gossiping is to ensure that eventually all correct processes can elect the same leader.

4.2 Correctness Proof

Given an execution, let C denote the set of processes that are correct in that execution. Let us consider the following set definitions (PL stands for "Potential Leaders"): $PL = \{p_x \mid \exists p_i \in C : count_i[x] \text{ is bounded}\}$, and for any correct process $p_i : PL_i = \{p_x \mid count_i[x] \text{ is bounded}\}$.

The proof is made up of three parts:
- We first show that, in any execution that satisfies MP, the set PL is not empty (Lemma 1),
- We then show that PL is a subset of the processes that are correct in that execution (Lemma 2),
- Finally, we show that $PL_i = PL$ for any correct process p_i (Lemma 3).

This first lemma shows that the additional assumption MP ensures that the set PL cannot be empty.

Lemma 1 $MP \Rightarrow PL \neq \emptyset$.

Proof Due to the MP assumption, there are a time t, a correct process p_i and a set Q including at least $f+1$ processes such that, after t, $\forall p_j \in Q$, until it possibly crashes, p_j receives from p_i only winning responses to its queries. Let us notice that Q includes at least one correct process.

Let us consider a time t' after which no more process crashes, and let $\tau = \max(t, t')$. Let p_k be any correct process. As p_k waits for RESPONSE() messages from $(n - f)$ processes and, after τ, at most $n - (f + 1)$ processes do not receive winning responses from p_i, it follows that there is a time τ_k, after which p_i always belongs to REC_FROM_k. From which we conclude that, after τ_k, p_k never increments $count_k[i]$ at line 105. As this is true for any correct process p_k, it follows that there is a time $\mathrm{T} \geq \max_{p_k \in C}(\tau_k)$ after which, due to the gossiping of the $count_k$ arrays, we have $count_{k1}[i] = count_{k2}[i] = M_i$ (a constant value), for any pair of correct processes p_{k1} and p_{k2}. The lemma follows.

$\square_{Lemma\ 1}$

The next corollary follows directly from the proof of the previous lemma.

Corollary 1 *Let p_i and p_j be any pair of correct processes. If, after some time, $count_i[k]$ remains forever equal to some constant value M_k, then there is a time after which $count_j[k]$ remains forever equal to the same value M_k.*

The second lemma shows that the set of potential leaders PL contains only correct processes.

Lemma 2 $PL \subseteq C$.

Proof We show the contrapositive, i.e., if p_x is a faulty process, then each correct process p_i is such that $count_i[x]$ increases forever. Thanks to the gossiping mechanism (realized by the QUERY_ALIVE() messages) used to periodically broadcast the counter arrays, it is actually sufficient to show that there is a correct process p_i such that $count_i[x]$ increases forever if p_x is faulty.

Let t_0 be a time after which all the faulty processes have crashed, and all the messages they have previously sent are received. Moreover, let $t > t_0$ be a time such that each correct process has issued and terminated a query-response between t_0 and t (the aim of this query-response invocation is to "clean up" -eliminate faulty processes from- the REC_FROM_i set of each correct process p_i). Let p_x be a faulty process (it crashed before t) and p_i be a correct process. We have the following:

- All the query-response invocations issued by p_i after t define a rec_from_i set (computed at line 106) that does not include p_x.
- It follows that, after t, the set REC_FROM_i computed at line 103 can never include p_x. This means that, after t, the set $not\_rec\_from_i$ (computed at line 104) always includes p_x. Hence, after t, $count_i[x]$ is increased each time p_i issues a query-response. As p_i is correct it never stops invoking the query-response mechanism, and the lemma follows.

$\square_{Lemma\ 2}$

Finally, the third lemma shows that no two processes can see different sets of potential leaders.

Lemma 3 $p_i \in C \Rightarrow PL_i = PL$.

[5] Actually, a timestamp-like pair $(count_i[\ell], \ell)$ is associated with each process p_ℓ. Then, when two counters have the same value, they are ordered according to their process ids.

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

```
     init: rec_from_i ← Π; count_i ← [0, ..., 0];

task T1:
     repeat
(101)   for_each j do send QUERY_ALIVE(count_i) to p_j end_do;
(102)   wait_until ( corresponding RESPONSE(rec_from) received from (n − f) proc. );
(103)   let REC_FROM_i = ∪ of all the rec_from_k received at line 102;
(104)   let not_rec_from_i = Π − REC_FROM_i;
(105)   for_each j ∈ not_rec_from_i do count_i[j] ← count_i[j] + 1 end_do;
(106)   let rec_from_i = the set of processes from which p_i received a RESPONSE at line 102
     end_repeat

task T2: upon reception of QUERY_ALIVE(c_j) from p_j:
(107)   for_each k ∈ Π do count_i[k] ← max(c_j[k], count_i[k]) end_do;
(108)   send RESPONSE(rec_from_i) to p_j

task T3: when leader() is invoked by the upper layer:
(109)   let ℓ such that (count_i[ℓ], ℓ) = min_{k∈Π}{(count_i[k], k)};
(110)   return (ℓ)
```

Figure 1. MP-based Module (for Process p_i)

Proof Let us first observe that $PL = \bigcup_{p_i \in C} PL_i$ (this follows immediately from the definition of PL). Consequently, $PL_i \subseteq PL$.

To show the inclusion in the other direction, let us consider $p_x \in PL$ (i.e., p_x is a correct process such that there is a correct p_j such that $count_j[x]$ is bounded). Let M_x be the greatest value taken by $count_j[x]$. We show that $count_i[x]$ is bounded. As after some time $count_j[x]$ remains forever equal to M_x, it follows from the fact that p_i and p_j are correct and the perpetual gossiping from p_i to p_j (lines 101 and 107) that we always have $count_i[x] \leq M_x$, from which we conclude that $count_i[x]$ is bounded. $\square_{Lemma\ 3}$

Theorem 2 *Let $1 \leq f < n$. The protocol described in Figure 1 implements a leader facility in $AS_{n,f}[MP]$.*

Proof The proof follows directly from the Lemmas 1, 2 and 3 which state that all the correct processes have the same non-empty set of potential leaders, which includes only correct processes. Moreover, due to Corollary 1, all the correct process have the same counter values for the processes of PL (and those values are the only ones to be bounded). It follows that the correct processes elect the same leader that is the correct process with the smallest counter value. $\square_{Theorem\ 2}$

Let us say "p_x is a process that makes satisfied the assertion MP" when p_x is a correct process such that after some time, there is a set of $f+1$ processes that receive only winning responses from it. It is important to notice that the process that is eventually elected is not necessarily a process p_x that makes satisfied the assertion MP.

4.3 Discussion

It is important to observe that query-response "challenges" issued by different processes are independent one from the other. This has an interesting consequence, namely, a process can introduce an arbitrary delay before issuing a query-response challenge (at line 101). Therefore, each process can, independendly of the other processes, dynamically define and set such a delay to match the bandwith that failure detector messages are allowed to use.

5 Leadership with Bounded Counters

The protocol presented in Figure 1 implements a leader facility as soon as the underlying system satisfies the behavioral property MP on the message exchange pattern. The common leader that is eventually elected is a correct process that satisfies some global "stability" property, namely, after some time no process suspects it. To attain this goal, the proposed protocol uses suspicion counters. Unfortunately, both the counters associated with the faulty processes and the counters associated with the correct processes that are not "potential leaders" increase forever. This section presents a protocol that requires only finite memory, namely, there is a single counter and this counter takes a finite number of values. This means that, in each infinite execution, both the local memory of each process and the message size are finite.

5.1 Underlying Principles

The protocol with finite memory is described in Figure 2. It borrows principles from sequence number-based protocols (similarly to what is done in [6]). There are two main

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

differences with respect to the previous "unbounded" protocol.

• A first difference lies in the way a process p_i represents the set of possible candidates for being the common leader. This is the role of the set $trust_i$. This set, built at line 204, provides a bounded representation of the set of potential leaders.

• The second difference lies in the use of sequence numbers that will remain finite in each execution. When a process broadcasts its current view of the set of potential leaders (line 201), it associates with it its "age", namely, the current value of $seqnum_i$. Let us notice that (as before) the query-response mechanism is used to convey this information. Then, when a process p_i receives a pair $(trust_j, seqnum_j)$ (task $T2$), it updates $trust_i$ according to the respective values of $seqnum_i$ and $seqnum_j$. If they are equal it trusts the processes in the intersection set (line 206). If it knows "less", it adopts the values it receives (207). As the new value of $trust_i$ can be computed from an intersection (at lines 204 and 207), it is possible that the new value be the empty set. In that case, p_i resets $trust_i$ to its initial value (namely, Π), and increments its sequence number to start a new age period.

Finally, the process that p_i considers as the leader is the process of $trust_i$ with the smallest identity.

5.2 Correctness Proof

Theorem 3 *Let $1 \leq f < n$. When run in $AS_{n,f}[MP]$, the protocol described in Figure 2 uses a finite memory and finite size messages.*

Proof Given an execution (with at least one correct process), let t_0 be a time after which (1) all the faulty processes have crashed and all their messages have been received, and (2) there is a correct process p_x and a set of $(f+1)$ processes that always receive winning responses from p_x (such a time t_0 exists due to the MP assumption). Finally, let $t > t_0$ be a time such that each correct process has invoked the query-response mechanism at least once between t_0 and t. (The idea is that after t the system has a "nice" behavior.)

Claim C1. After t, there is a correct process p_x that continuously belongs to the REC_FROM_i set of every correct process p_i.

Proof of the claim. Let p_i be any correct process. As (1) p_i waits for RESPONSE() messages from $(n-f)$ processes and (2) due to the MP assumption, after t, at most $n-(f+1)$ processes do not receive winning responses from some p_x, it follows that p_i receives at least one rec_from_k set including p_x. The claim follows then from line 203: after t, p_x always belongs to REC_FROM_k. *End of the proof of the claim C1.*

Let M^t be the maximal $seqnum$ value among the correct processes at time t. Moreover, let say "the set $trusted$ is associated with the sequence number sn" when there is a correct process p_j such that $trust_j = trusted$ and $seqnum_j = sn$ (let us observe that several sets can be associated with the same sequence number).

Claim C2. Let us assume that $\emptyset$ is associated with M^t. There is then (1) a process p_j that executes the reset statement at line 208, after which we have $(trust_j, seqnum_j) = (\Pi, M^t+1)$. Moreover, (2) the pair (Π, M^t+1) is sent to all the processes.

Proof of the claim. Let us first observe that (Observation $O1$) a set $trust_i$ can only decrease while $seqnum_i$ remains equal to M^t, (Observation $O2$) there is no gap in sequence numbers (which means that if a sequence number variable is equal to M, then there are sequence number variables that had previously the values $0, 1, \ldots, M-1$), and (Observation $O3$) the update by a process p_j of its $seqnum_j$ variable to the value $M+1$ (at line 207 or 208) is always due to the fact that some process p_k (which is possibly p_j itself) executed $seqnum_k \leftarrow seqnum_k + 1$ at line 208 (where M is the value of $seqnum_k$ before the update; notice that p_k also set $trust_k$ to Π).

Let p_i be a process that associates $\emptyset$ with M^t. If the pair $(trust_i, seqnum_i)$ remains equal to $(\emptyset, M^t)$ until p_i receives a query, it executes line 208 and consequently resets $(trust_i, seqnum_i)$ to (Π, M^t+1). The only other possibility for that pair to be modified is at line 207, but in that case p_i received a sequence number $> M^t$, and it follows from the observations $O2$ and $O3$ that some process p_j executed line 208 updating the pair $(trust_j, seqnum_j)$ to (Π, M^t+1). This proves the first part of the claim.

The proof of the second part of the claim is by contradiction. Let us assume that no process issues a query with the pair (Π, M^t+1). This means that the pairs that are sent have the form (X, M^t+1) with $X \neq \Pi$. Let p_{i1} be a process such that at some time t_{i1} we have $(trust_{i1}, seqnum_{i1}) = (\Pi, M^t+1)$. As it sends at some time $t'_{i1} > t_{i1}$ the pair (X_{i1}, M^t+1) with $X_{i1} \neq \Pi$ (by assumption), we conclude that, between t_{i1} and t'_{i1}, p_{i1} received a query from some process p_{i2} carrying (X_{i2}, M^t+1) with $X_{i2} \neq \Pi$. The same reasoning can be applied to p_{i2}, from which we conclude that there is a process p_{i3}, etc. It follows that we can construct an infinite chain of distinct processes, which is clearly impossible as there is a finite number of processes. It follows that there is a process that sends a query carrying the pair (Π, M^t+1). *End of the proof of the claim C2.*

Let p_i be a correct process such that $seqnum_i = M^t$. Due to the gossiping mechanism, after some time we will have $seqnum_j \geq M^t$ for any correct process p_j. We con-

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

```
init: rec_from_i ← Π; seqnum_i ← 0; trust_i ← Π;

task T1:
     repeat
(201) for_each j do send QUERY_ALIVE(trust_i, seqnum_i) to p_j end_do;
(202) wait_until ( corresponding RESPONSE(rec_from) received from (n − f) proc. );
(203) let REC_FROM_i = ∪ of all the rec_from_k received at line 202;
(204) trust_i ← trust_i ∩ REC_FROM_i;
(205) let rec_from_i = the set of processes from which p_i received a RESPONSE at line 102
     end_repeat

task T2: upon reception of QUERY_ALIVE(trust_j, seqnum_j) from p_j:
(206) if seqnum_j = seqnum_i then trust_i ← trust_i ∩ trust_j end_if;
(207) if seqnum_j > seqnum_i then trust_i ← trust_j; seqnum_i ← seqnum_j end_if;
(208) if trust_i = ∅ then trust_i ← Π; seqnum_i ← seqnum_i + 1 end_if;
(209) send RESPONSE(rec_from_i) to p_j

task T3: when leader() is invoked by the upper layer:
(210) if trust_i = ∅ then return (i)
(211)                 else return (min(trust_i))
(212) end_if
```

Figure 2. MP-based Module with Finite Memory (for Process p_i)

sider two cases.

• Case 1: $\emptyset$ is never associated with M^t. In that case, no correct process p_i will ever execute the reset statement at line 208. It follows that no process p_i will increase its $seqnum_i$ variable, and the theorem follows.

•Case 2: $\emptyset$ is associated with M^t. From the claim ***C2***, there is inevitably a process p_j that executes the reset statement at line 208, after which we have $(trust_j, seqnum_j) = (\Pi, M^t + 1)$, and this pair is sent to all the correct processes. Combining this with the claim ***C1***, from now on, any set $trust_i$ permanently contains at least the correct process p_x defined in ***C1***. It follows that a set $trust_i$ can become empty neither at line 204 nor at line 206. Hence, no process p_i can execute the reset statement at line 208, and the theorem follows. $\square_{Theorem\ 3}$

Theorem 4 *Let $1 \leq f < n$. The protocol described in Figure 2 implements a leader facility in $AS_{n,f}[MP]$.*

Proof Given an execution and C being the set of processes that are correct in that execution, let: $PL = \{p_x \mid \exists p_i \in C :$ after some time x remains continuously in $trust_i\}$.

We first show that $MP \Rightarrow PL \neq \emptyset$. This is a consequence of Theorem 3 (which relies on the MP assumption). More precisely, there is a time t after which the sequence numbers do no longer increase, from which we conclude that that no set $trust_i$ becomes empty after t. Moreover, the gossiping mechanism ensures that there is a time $t' \geq t$ after which all these sets are equal and do not change their value (as they are then updated only by intersection). Finally, due to the very definition of PL, the $trust_i$ sets are then equal to PL. It follows that PL is not empty.

We now show that $PL \subseteq C$. This actually follows from the claim ***C1*** stated and proved in Theorem 3. More explicitly, as there is a time after which the REC_FROM_i sets contain only correct processes and the $trust_i$ sets are never reset to Π, it follows that, after that time, these $trust_i$ sets can contain only correct processes.

Finally, the eventual leadership property follows from the following observations: $PL \neq \emptyset$, $PL \subseteq C$, and there is a time t after which we have $\forall i :\ p_i \in C \Rightarrow trust_i = PL$. $\square_{Theorem\ 4}$

6 Conclusion

Leader-based protocols are common in distributed computing. They rely on an underlying primitive that provide the same unique leader to the processes. Such a primitive is usually used to solve synchronization or coordination problems. While it is particularly easy to implement a leader primitive in a fault-free system, its construction in an asynchronous system prone to process crashes is impossible if the underlying system is not enriched with additional assumptions. While the traditional approach to build a distributed leader facility in such crash-prone asynchronous systems considers additional synchrony assumptions, the approach presented in this paper has considered an additional time-free assumption, namely, a behavioral property on the message flow.

The paper has presented two leader protocols. The first one uses potentially unbounded counters, while the second one (which is a little bit more involved) requires only finite memory. Interestingly, as in [17], it is possible to merge the

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

synchrony-based approach with the the proposed approach to get a hybrid leader protocol. Such a combination is described in the appendix. It allows expediting the convergence (a correct process is elected as the definitive leader) as, in that case, convergence can then be guaranteed as soon as one assumption (synchrony or message exchange pattern) is satisfied. Such a hybrid approach provides increased overall assumption coverage [21].

Acknowledgements

We would like to thank X. Defago and D. Powell for (independent) discussions on the implementation of failure detectors, and the referees for their constructive comments.

References

[1] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Omega with Weak Reliability and Synchrony Assumptions. *22th ACM Symposium on Principles of Distributed Computing*, pp. 306-314, 2003.

[2] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication Efficient Leader Election and Consensus with Lilmited Link Synchrony. *23th ACM Symposium on Principles of Distributed Computing*, 2004.

[3] Anceaume E., Fernandez A., Mostefaoui A., Neiger G. and Raynal M., Necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors. *Journal of Computer and System Sciences,* 68:123-133, 2004.

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

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

[6] Chu F., Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.

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

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

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

[10] Guerraoui R., Indulgent Algorithms. *19th ACM Symposium on Principles of Distributed Computing, (PODC'00)*, ACM Press, pp. 289-298, 2000.

[11] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers,* 53(4):453-466, April 2004.

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

[13] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[14] Larrea M., Fernández A. and Arèvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Reliable Distributed Systems*, pp. 52-60, 2000.

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

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

[17] Mostefaoui A., Powell D., and Raynal M., A Hybrid Approach for Building Eventually Accurate Failure Detectors. *Proc. 10th IEEE Int. Pacific Rim Dependable Computing Symposium (PRDC'04)*, pp. 57-65, 2004.

[18] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symp. on DIStributed Computing*, Springer Verlag LNCS #1693, pp. 49-63, 1999.

[19] Mostefaoui A. and Raynal M., Low-Cost Consensus-Based Atomic Broadcast. *7th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'2000)*, pp. 45-52, 2000.

[20] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

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

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

A A Hybrid Protocol

This appendix shows that the previous approach (based on a property satisfied by the message exchange pattern) and the more classical approach that relies on the use of timeouts are not antagonistic and can be combined to produce a hybrid protocol implementing an eventual leader oracle. The resulting protocol benefits from the best of both worlds in that it converges as soon as some synchrony assumption is satisfied, or the required message exchange pattern occurs. We consider here the synchrony assumption and the corresponding leader protocol defined in [2].

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE

A.1 Synchrony Assumptions

We consider here a synchrony model slightly stronger than the one introduced in [2][6]. First, the processes are synchronous (there is a lower and upper bound on the number of steps per time unit of any non-faulty process). Moreover, there is at least one correct process that is a $\diamond f$-source. This means that there is a correct process p_i that has f output channels such that, after some unknown but finite time t, there is a bound δ (whose value is not known in advance) such that -after t- any message sent on such a channel is received within δ (it is not required that the destination processes of a $\diamond f$-source be correct; some -or all- of them can be faulty[7]).

Let $AS_{n,f}[\diamond f\text{-source}]$ denote a distributed system satisfying this synchrony assumption.

A.2 Aguilera *et al.*'s Protocol

Aguilera *et al.* present in [2] the following leader protocol, whose code for process p_i is described in Figure 3, that works in any system $AS_{n,f}[\diamond f\text{-source}]$. Each process p_i manages an array $count_i$. This array is such that $count_i[j]$ is bounded if p_j is a correct $\diamond f$-source, while $count_i[j]$ is unbounded if p_j is faulty. As in the protocol described in Figure 1, the leader is the process p_ℓ whose counter has the smallest value (task $T5$).

The key of the protocol is the management of each counter $count_i[j]$, i.e., the way such a counter is (or not) increased. To this end, each process p_i manages an array $suspect_i$ as follows (task $T4$): $suspect_i[j]$ keeps track of the set of processes that currently suspect p_j to have crashed(task $T3$). If this set contains $(n-f)$ processes (or more), then p_i increases $count_i[j]$ and resets $suspect_i[j]$ to $\emptyset$. The $\diamond f$-source assumption allows showing that every faulty process p_j will be forever suspected (i.e., $count_i[j]$ will never stop increasing), while $count_i[j]$ will remain bounded if p_j is a $\diamond$ f-source. Consequently, there is at least one entry of $count_i$ that remains bounded and all the entries of $count_i$ that remain bounded correspond to correct processes. So, we get the following theorem:

Theorem 5 [2] *The protocol in Figure 3 implements a leader facility in* $AS_{n,f}[\diamond f\text{-source}]$.

A.3 A Hybrid Protocol

Let $count\_MP_i$ be the array $count_i$ used in the protocol described in Figure 1 (the protocol based on the message exchange pattern assumption). Similarly, let $count\_\diamond f_i$ be the array $count_i$ used in the protocol described in Figure 3 (the protocol based on the $\diamond f$-source synchrony assumption).

[6]While [2] considers that the channels can be fair lossy, we consider here that they are reliable.

[7]This is similar to the MP assumption where some of the $f+1$ processes of the set Q can be faulty.

init:
$\forall j \neq i$: $timeout_i[j] \leftarrow \alpha + 1$; set $timer_i[j]$ to $timeout_i[j]$;
$count_i \leftarrow [0, \ldots, 0]$; $suspect_i \leftarrow [\emptyset, \ldots, \emptyset]$

task $T1$: **repeat periodically every** α **time units**:
for_each $j \neq i$ **do** *send* ALIVE $(count_i)$ *to* p_j **end_do**

task $T2$: **when** ALIVE $(count)$ **is received from** p_j:
$\forall k$ **do** $count_i[k] \leftarrow \max(count_i[k], count[k])$ **enddo**;
reset $timer_i[j]$ to $timeout_i[j]$

task $T3$: **when** $timer_i[k]$ **expires**:
$timeout_i[k] \leftarrow timeout_i[k] + 1$;
$\forall j$ **do** *send* SUSPECT(k) *to* p_j **enddo**;
reset $timer_i[k]$ to $timeout_i[k]$

task $T4$: **when** SUSPECT(k) **is received from** p_j:
$suspect_i[k] \leftarrow suspect_i[k] \cup \{p_j\}$;
if $|suspect_i[k]| \geq n - f$ **then**
$count_i[k] \leftarrow count_i[k] + 1$; $suspect_i[k] \leftarrow \emptyset$
end if

task $T5$: **when** leader() **is invoked by the upper layer**:
let ℓ **s. t.** $(count_i[\ell], \ell) = min_{k \in \Pi}\{(count_i[k], k)\}$;
return (ℓ)

Figure 3. Aguilera *et al.*'s Leader Protocol

These protocols can be merged as follows. Both protocols execute independently one from the other with the following modification. The last task of each protocol (i.e., the task $T3$ in Figure 1, and the task $T5$ in Figure 3) are suppressed and replaced by a new task $T3/T5$ defined as follows:

task $T3/5$: **when** leader() **is invoked by the upper layer**:
$\forall k$: $count_i[k] \leftarrow \min(count\_MP_i[k], count\_\diamond f_i[k])$;
let ℓ **s. t.** $(count_i[\ell], \ell) = min_{k \in \Pi}\big\{(count_i[k], k)\big\}$;
return (ℓ)

The reader can adapt the proof described in the paper to show that the resulting hybrid protocol implements a leader facility as soon as either the message pattern assumption MP or the $\diamond f$-source synchrony assumption is satisfied. So we get the following theorem:

Theorem 6 *The hybrid protocol obtained by combining the protocol described in Figure 1 and the protocol described in Figure 3 implements a leader facility in* $AS_{n,f}[MP \vee \diamond f\text{-source}]$.

Hence, this protocol benefits from the best of both worlds. This shows that, when the underlying system can satisfy several alternative assumptions, convergence can be expedited. Moreover, since convergence is guaranteed if any one of the alternative assumptions is satisfied, the resulting hybrid protocol provides an increased overall assumption coverage [21].

Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)
1060-9857/04 $ 20.00 IEEE