Parallel Processing Letters,

A TIME-FREE ASSUMPTION TO IMPLEMENT EVENTUAL LEADERSHIP

ACHOUR MOSTEFAOUI
IRISA, Université de Rennes, Campus de Beaulieu, France

and

ERIC MOURGAYA
IRISA, Université de Rennes, Campus de Beaulieu, France

and

MICHEL RAYNAL
IRISA, Université de Rennes, Campus de Beaulieu, France

and

CORENTIN TRAVERS
IRISA, Université de Rennes, Campus de Beaulieu, France

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

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 introduces a novel approach to implement an eventual leader protocol, namely a *time-free* behavioral assumption on the flow of messages that are exchanged. It presents a very simple leader protocol based on this assumption. It then presents a second leader protocol combining this timeless assumption with eventually timely channels. As it considers several assumptions, the resulting hybrid protocol has the noteworthy feature to provide an increased overall assumption coverage. A probabilistic analysis shows that the time-free assumption is practically always satisfied.

Keywords: Asynchronous system, Distributed algorithm, Fault tolerance, Hybrid Algorithm, Leader election, Process crash, Time-free Protocol, Timely Channel.

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 [8].

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 process delivers at least the messages sent by the correct processes (a *correct* process is a process that does not crash). 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 [14]. To overcome this major difficulty, Chandra and Toueg have introduced the concept of *Unreliable Failure Detector* [4]. A failure detector is a device that outputs failure related information according to the current failure pattern. Among all the classes of failure detectors, we are interested here in the class of failure detectors denoted Ω, and called the class of *eventual leader* oracles. Such an oracle offers a primitive leader() 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) [5,18]. It has also been shown that, as far as failure detection is concerned, Ω is the weakest failure detector class that allows solving consensus [5]. Ω is the oracle that is (implicitly) used to ensure the termination property in Lamport's Paxos protocol [12].

Unfortunately, Ω cannot be implemented in pure (time-free) asynchronous systems (its implementation would contradict the consensus impossibility result [8]). Nevertheless, such an oracle allows the protocols that use it to benefit from a very nice property, namely *indulgence* [9,10]. More precisely, 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, Ω defines a class of oracles that allow the design of indulgent consensus protocols [10]. 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 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 of an 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 Ω-based protocols, and motivate the design of underlying "best effort" protocols that implement an 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 Ω (e.g., [1,7,13]). 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 [6]. 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 [15]. 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 implementing some classes of failure detectors.

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 MP be the following behavioral property on the query/response exchange pattern (MP stands for *Message Pattern*): "There 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 [15,16,19] that some failure detector classes can be implemented when this property is satisfied.

The 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 first protocol presented in this paper shows 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 also a second protocol, namely a hybrid leader protocol that benefits from the best of both worlds: it elects a leader as soon as MP is satisfied or some channels are eventually timely. In that sense, this protocol is practically appealing as it can provide a better assumption coverage [20] than a leader protocol based on a single property.

2. 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 ($1 \leq f < n$). This means that $1 \leq f < n$ is an assumption on the system behavior. More precisely, in all the executions where at most f processes crash, the upper layer protocol we are interested in has to work correctly (eventually elect a common leader). On the contrary, in the executions where more than f processes

crash, there is no guarantee on the the upper layer protocol (as we will see, if more than f processes crash, the proposed leader protocol can block).

Processes communicate 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).

A process p_i has local variables, and consists of one or several local tasks. When a process is made up of several local tasks, it is implicitly assumed that these tasks access the local variables in mutual exclusion. So, the local variables have the *atomicity* semantics with respect to the local tasks.

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).

Both a QUERY_ALIVE() message and a RESPONSE() message can be used to piggyback values. This allows the querying process to disseminate a value to all the processes, and to obtain a value from each process.

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. It is assumed that (until it possibly crash) a process p_i issues forever sequential queries.

Figure 1 depicts a query-response mechanism in a system made up of $n = 6$ processes, and $f = 2$. After p_3 broadcasts QUERY_ALIVE(), the $n - f = 4$ first RESPONSE() messages it receives are from p_2, p_3 (itself), p_5 and p_6. These responses are the winning responses for that query. Notice that p_4 has crashed.

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.

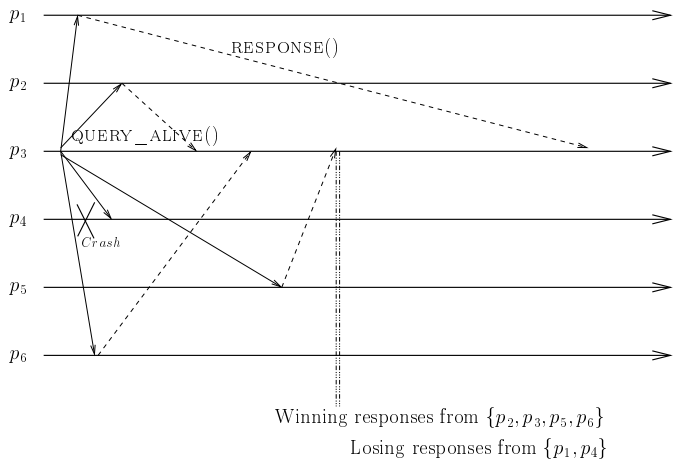


Figure 1: Query/Response Mechanism

3. 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* (that, as already indicated, is a shortcut for *Message Pattern*):

> "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 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 [15] 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, and satisfying the property *MP*. The appendix studies the case $f = 1$ and shows that *MP* is then practically always satisfied.

4. 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 (his property refers to a notion of global time that is not accessible to the processes):

- 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 [10,12,18]*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., [4,12,17]).

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 [8], it follows that a leader oracle cannot be implemented in an asynchronous system $AS_{n,f}[\emptyset]$ with $f < n/2$. 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 $f < n$.*

Proof (This proof is close to the proof we give in [3] where we show that there is no protocol implementing an eventually weak failure detector in $AS_{n,f}[\emptyset]$ with $f < n$.) 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).

*The Paxos protocol [12] 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.)

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 $t_1 < t_2 < t_3 < \cdots$ and such that no two consecutive leaders are the same process. Hence, the eventual leadership property we have assumed is not satisfied. $\square_{Theorem\ 1}$

5. An *MP*-based Asynchronous Leader Protocol

5.1. 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 less. 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 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 the $count_i$ array, the role of the Task $T3$ of p_i is to define its current leader. More explicitly, the current leader of p_i is the process p_ℓ such that the pair $(\ell, count_i[\ell])$ is the smallest pair over all the pairs $(x, count_i[x])$, for $1 \leq x \leq n$. This is the classical lexicographical order, namely, $(\ell_1, c_1) < (\ell_2, c_2)$ if $(c_1 < c_2)$ or $(c1 = c_2 \wedge \ell_1 \leq \ell_2)$.

The aim of the tasks $T1$ 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 1-2). 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.

Let us observe that all the tasks that define p_i access a single local variable, namely, $count_i[1..n]$. It follows that they cannot deadlock (assuming a deadlock-free underlying locking mechanism).

```
     init: rec_from_i ← Π; count_i ← [0, ..., 0];
     Launch in parallel the tasks T1, T2 and T3 (they access count_i in mutual exclusion)

task T1:
     repeat
 (1)     for_each j do send QUERY_ALIVE(count_i) to p_j end_do;
 (2)     wait_until ( corresponding RESPONSE(rec_from) received from (n − f) proc. );
 (3)     let REC_FROM_i = ∪ of all the rec_from_k received at line 2;
 (4)     let not_rec_from_i = Π − REC_FROM_i;
 (5)     for_each j ∈ not_rec_from_i do count_i[j] ← count_i[j] + 1 end_do;
 (6)     let rec_from_i = the set of processes from which p_i received a RESPONSE at line 2
     end_repeat

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

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

Figure 2: *MP*-based Module (for Process p_i)

5.2. 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"):

$$PL = \{p_x \mid \exists p_i \in C : \ count_i[x] \text{ is bounded}\},$$

$$\text{For any correct process } p_i : \quad PL_i = \{p_x \mid count_i[x] \text{ is bounded}\}.$$

The proof is decomposed into four lemmas. It assumes that at most f processes crash, and that a process regularly issues queries (until it possibly crashes).

The 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 5. As this is true for any correct process p_x, it follows that there is a time $\mathrm{T} \geq \max_{p_x \in C}(\tau_x)$ after which, due to the gossiping of the $count_x$ arrays, we have $count_{x1}[i] = count_{x2}[i] = M_i$ (a constant value), for any pair of correct processes p_{x1} and p_{x2}. The lemma follows. $\square_{Lemma\ 1}$

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 sent by every 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 6) that does not include p_x.
- It follows that, after t, the set REC_FROM_i computed at line 3 can never include p_x. This means that, after t, the set $not\_rec\_from_i$ (computed at line 4) 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 $p_i \in C \Rightarrow PL_i = PL$.

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 1 and 7) that we always have $count_i[x] \leq M_x$, from which we conclude that $count_i[x]$ is bounded. $\square_{Lemma\ 3}$

The last lemma shows that, due to the gossiping mechanism, if a counter value remains bounded, it eventually takes the same value at all the correct processes.

Lemma 4 *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.*

Proof The proof is by contradiction. let us assume that (1) p_i and p_j are correct, (2) from some time t_1, $count_i[k]$ stays constant at the value M_k, and (3) there is a time $t_2 > t_1$ after which $count_j[k]$ becomes equal to $M'_k > M_k$.

As p_j is correct, it sends query messages carrying the value $count_j[k] = M'_k$ after time t_2. As p_i is correct, it receives such a message and updates consequently $count_i[k]$ to $M'_k > M_k$, contradicting the assumption stating that, after t_1, $count_i[k]$ remains forever equal to M_k. $\square_{Lemma\ 4}$

Theorem 2 *The protocol described in Figure 2 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 Lemma 4, 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. $\square_{Theorem\ 2}$

This protocol uses unbounded counters. This property can be useful for some classes of applications (e.g., [11]). It is nevertheless possible to obtain a MP-based leader protocol that uses only finite memory [19].

6. A Hybrid Protocol

This section 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].

6.1. Synchrony Assumptions

We consider here a synchrony model slightly stronger than the one introduced in [2][†] 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

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

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).

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

6.2. Aguilera et al.'s Protocol

Aguilera *et al.* present in [2] the following leader protocol, 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 2, the leader is the process p_ℓ whose counter has the smallest value (task $T5$).

init: **for_each** $j \neq i$ **do** $timeout_i[j] \leftarrow \alpha + 1$; set $timer_i[j]$ to $timeout_i[j]$ **end_do**;
$count_i \leftarrow [0, \ldots, 0]$; $suspect_i \leftarrow [\emptyset, \ldots, \emptyset]$;
Launch in parallel the tasks $T1$ until $T5$

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:
for_each 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**:
$timeout_i[k] \leftarrow timeout_i[k] + 1$;
for_each 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 **such that** $(count_i[\ell], \ell) = min_{k \in \Pi}\{(count_i[k], k)\}$;
return (ℓ)

Figure 3: Aguilera *et al.*'s Leader Protocol (for Process p_i)

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 $\emptyset$. It is relatively easy to see that if a process p_j crashes, then $count_i[j]$ will never stop increasing at any

non-crashed process p_i. The $\diamond f$-source assumption allows showing that $count_i[j]$ 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 3 [2] *The protocol in Figure 3 implements a leader facility in* $AS_{n,f}[\diamond f$-source$]$.

6.3. A Hybrid Protocol

Let $count\_MP_i$ be the array $count_i$ used in the protocol described in Figure 2 (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).

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 2, and the task $T5$ in Figure 3) are suppressed and replaced by a new task $T3/T5$ defined as follows:

task $T3/T5$: **when** leader() **is invoked by the upper layer**:
for_each k **do** $count_i[k] \leftarrow \min(count\_MP_i[k], count\_\diamond f_i[k])$ **enddo**;
let ℓ **such that** $(count_i[\ell], \ell) = min_{k \in \Pi}\{(count_i[k], k)\}$;
return (ℓ)

The previous proof can be easily adapted 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 4 *The hybrid protocol obtained by combining the protocol described in Figure 2 and the protocol described in Figure 3 implements a leader facility in* $AS_{n,f}[MP \vee \diamond f$-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 [20].

7. Conclusion

Leader-based protocols are common in distributed computing. They rely on an underlying primitive that eventually provides the processes with the same unique leader. 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 is based on a property on the message exchange pattern generated by query and response messages. The second merges the synchrony-based approach with the the proposed approach to get a hybrid leader protocol. This protocol 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, thereby providing an increased overall assumption coverage.

Acknowledgments

We would like to thank Kemal Ebcioğlu (the managing editor of this special issue) and the referees for their constructive comments that helped improve the presentation of the paper.

References

[1] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Omega with Weak Reliability and Synchrony Assumptions. *Proc. 22th ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 306-314, Boston (MA), 2003.

[2] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337, St. John's, Newfoundland (Canada), 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] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.

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

[8] 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.

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

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

[11] Hayashibara N., Defago X., Yared Y. and Katayama T., The ϕ Accrual Failure Detector. *Proc. 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 66-78, Florianpolis (Brasil), 2004.

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

[13] Larrea M., Fernndez A. and Arvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Society Press, pp. 52-60, Nuremberg (Germany), 2000.

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

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

[16] 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)*, IEEE Computer Society Press, pp. 57-65, Papeete, (Tahiti, France), 2004.

[17] Mostefaoui A. and Raynal M., Low-Cost Consensus-Based Atomic Broadcast. *7th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'2000)*, IEEE Computer Society Press, UCLA, Los Angeles (CA), pp. 45-52, 2000.

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

[19] Mostefaoui A., Raynal M. and Travers C., Crash-Resilient Time-free Eventual Leadership. *Proc. 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 208-217, Florianpolis (Brasil), 2004.

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

Appendix A

A.1. The Case $f = 1$

A.1.1. A Channel Property

We can assume that the response from a process to its own queries always arrive among the first $(n - f)$ responses it is waiting for. Let $MP(1)$ denote the property MP when $f = 1$. $MP(1)$ means that there is a set of two processes, say $\{p_i, p_j\}$, such that p_j (until it possibly crashes) always receives and processes the response from p_i to its queries (in other words, the responses from p_i always arrive among the $(n - 1)$ responses p_j is waiting for and so are never discarded by p_j).

Let a query be *terminated* when the corresponding process has received all the corresponding winning responses. Considering the last query issued by p_k terminated at or before t, let $rtd(k, \ell, t)$ denote the round-trip delay of the corresponding query-response exchanged between p_k and p_ℓ. If there is no response from p_ℓ or if the response is discarded by p_k, let $rtd(k, \ell, t) = +\infty$. With these notations $MP(1)$

an be rewritten as follows when

$$\begin{aligned} MP(1) \quad \equiv \quad & \exists \text{ a time } t',\ \exists (p_i, p_j) \text{ such that} \\ & \forall t > t' : (p_j \text{ not crashed at } t) \Rightarrow \big(rtd(i,j,t) \neq +\infty\big). \end{aligned}$$

This property can be rephrased as a property on the behavior of the channels, namely: *"There is a time after which there is a channel in the system, say (p_i, p_j), that is never the slowest among the channels connecting p_i or p_j to the other processes"*. It follows that, when the underlying system satisfies this channel property, the protocol described in Figure 2 builds a failure detector of the class Ω despite one process crash.

A.2. Probabilistic Analysis

This section computes the probability that the property $MP(1)$ be satisfied from the very beginning in an asynchronous distributed system made up of n processes in which at most one process can crash, i.e., to compute

$$\text{Prob}\big[\exists (p_i, p_j) : \forall t : (p_j \text{ not crashed at } t) \Rightarrow \big(rtd(i,j,t) \neq +\infty\big)\big].$$

To compute such a probability we assume that no process crashes (this is because, as soon as a process crashes, $MP(1)$ is trivially satisfied). So, assuming no process crash, we want to compute:

$$\text{Prob}\big[\exists (p_i, p_j) : \forall t : \big(rtd(i,j,t) \neq +\infty\big)\big].$$

In order to take into account the time parameter t in a simple way, we consider that a protocol execution proceeds in consecutive asynchronous rounds in the sense that during each "round" r, each process issues a query, waits for the first $(n-1)$ corresponding responses, and then proceeds to the next "round". As the system is asynchronous, the $(n-1)$ RESPONSE messages associated with a query issued by a process p_i can arrive to it in any order. It follows that the previous probability tends towards 0 when t (i.e., r) tends towards $+\infty$ (this follows from the fact that the system is asynchronous). But, as no real execution is infinite, we are interested in protocol executions made up of a finite number x of rounds. Actually, x measures the length of a time period during which the failure detector is used. So, from a realistic point of view, we are interested in computing the following probability $p(x)$:

$$p(x) = \text{Prob}\big[\exists (p_i, p_j) : \forall r \leq x : \big(rtd(j,i,r) \neq +\infty\big)\big].$$

Let $discarded_i(r)$ be the identity of the process whose response has not been received among the $(n-1)$ first responses to the query issued by p_i during round r. As by assumption no process crashes, $discarded_i(r)$ is always defined, and we have $j = discarded_i(r) \Leftrightarrow rtd(i,j,r) = +\infty$. Hence, $p(x)$ can be rewritten as follows:

$$p(x) = \text{Prob}\big[\exists p_i, \exists p_j \neq p_i : \bigwedge_{0<r\leq x} (j \neq discarded_i(r))\big].$$

Computing the $p(x)$ function

Let $prop(i,x) \equiv \forall p_j \neq p_i : \bigvee_{0<r\leq x} (j = discarded_i(r))]$. We have:

$$p(x) = \text{Prob}[\exists p_i : \neg\, prop(i,x)],$$

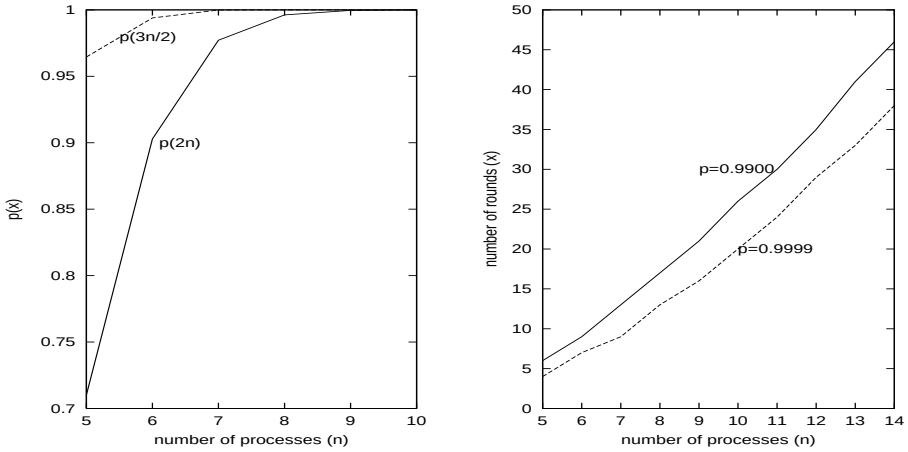
i.e.,

$$p(x) = 1 - \text{Prob}[\forall p_i : prop(i,x)].$$

Let $pp(i,x) = \text{Prob}[prop(i,x)]$. As no process plays a particular role, for any (i,j,x) we have $pp(i,x) = pp(j,x)$. Let $pp(x)$ be that value. Moreover, $\forall(i,j,r,r')$ the identities $k = discarded_i(r)$ and $k' = discarded_j(r')$ are independent random variables (they are not related by the query mechanism). Moreover, the property $prop(i,x)$ is verified (or not) for each process p_i independently from the other processes. We can conclude from this discussion that $p(x)$ follows a binomial distribution and we get:

$$p(x) = 1 - \big(pp(x)\big)^n.$$

The computation of $pp(x)$ can be done by observing that the x (probabilistic) events corresponding to the collect of the $(n-1)$ response messages do correspond to a "trial with replacement" among $(n-1)$ values (as already indicated, we assume that a process p_i always receives its own responses to its own queries). Each query can be seen as "selecting" a process id to discard it, and so $pp(x)$ is the probability that each process identity is drawn out at least once. It follows that $pp(x)$ can be computed from basic probability theorems. Then, as soon as $pp(x)$ is determined, one can easily compute $p(x)$.



(a) $p(x)$ according to n for $x = 3n/2$ and $x = 2n$

(b) Fixing a priori a probability p

Figure A.1: Measuring "how realistic" is *MP(1)*

Practical results The previous determination of the probability $p(x)$ can be used to evaluate "how realistic" is the property $MP(1)$ in an asynchronous system. Let us notice that the number of rounds considered (x) can be interpreted as the duration of a session during which the upper layer application uses the leader oracle.

We have seen that $p(x)$ tends towards 0 when x tends towards $+\infty$. So, it is interesting to know if $p(x)$ tends quickly towards 0 or not. To this aim, let us consider the two curves depicted in Figure A.1.(a) whose horizontal axis corresponds to the number n of processes, and vertical axis corresponds to the probability $p(x)$. The curves correspond to the cases where the number x of rounds that are considered are $x = 3n/2$ and $x = 2n$, respectively (the highest curve depicts $p(3n/2)$, while the lowest one depicts $p(2n)$, for $5 \leq n \leq 10$). These curves first show how $p(x)$ is related to the session length x. More interestingly, they also show that, when $x \leq 2n$, $p(x)$ is very close to 1 for $n \geq 7$. Intuitively, adding processes can only create more situations where $MP(1)$ is satisfied. This is confirmed and measured in Figure A.1.(a) that shows that, when the number n of processes increases, the probability increases also and becomes very quickly very close to 1. In all cases, from $n = 7$, the probability $p(x)$ is very close to 1. This means that $MP(1)$ is practically satisfied in asynchronous distributed systems made up of $n \geq 7$ processes, if the sessions (during which the upper layer application uses the leader oracle) are not "too long".

Figure A.1.(b) provides a complementary view. Its horizontal axis corresponds to the number n of processes, while its vertical axis corresponds to the length of the observation period (number x of rounds). Considering a given probability p, this figure shows how long an upper layer session can be (number x of rounds) for an asynchronous system made up of n processes ($5 \leq n \leq 14$) to satisfy $MP(1)$ with the given probability. The lowest curve corresponds to $p = 0.9999$, the highest corresponds to $p = 0.9900$. These curves show in another way that an asynchronous distributed system satisfies $MP(1)$ with a very high probability.