A Timing Assumption and a t-Resilient Protocol for Implementing an Eventual Leader Service in Asynchronous Shared Memory Systems

Antonio FERNÁNDEZ[†] Ernesto JIMÉNEZ[‡] Michel RAYNAL[⋆] Gilles TRÉDAN[⋆]

[†] LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain

[‡] EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain

[⋆] IRISA, Université de Rennes, Campus de Beaulieu 35 042 Rennes, France

anto@gsyc.escet.urjc.es ernes@eui.upm.es raynal@irisa.fr gtredan@irisa.fr

Abstract

While electing an eventual common leader, despite process crashes, in a shared memory system where the processes communicate only by reading and writing shared registers is possible when the processes progress synchronously, this problem becomes impossible to solve as soon as the processes can progress in a fully asynchronous way. So, an important problem consists in finding additional behavioral assumptions that are, at the same time, "as weak as possible" (in order they are practically always satisfied), and "strong enough" in order to allow implementing an eventual leader service despite the net effect of asynchrony and failures. This paper focuses on this dilemma. More explicitly, it investigates a timing assumption that allows implementing an eventual leader in presence of partial asynchrony and process crashes. The proposed timing assumptions are particularly weak. They are the following: after some time (i) there is a process that behaves synchronously, and (ii) $(t - f)$ other processes have timers that work correctly (t is the maximal number of processes that may crash, and f the actual number of process crashes; a timer works incorrectly when it expires too early with respect to the value it has been set). Then, the paper proposes a t-resilient protocol that elects an eventual common leader in any shared memory system that satisfies the previous assumption. Interestingly, this protocol is based on simple design principles.

Keywords: *Asynchronous system, Atomic register, Eventual leader, Fault-tolerance, Omega, Process crash, Shared memory, System model, Timer property, Timing assumptions.*

1 Introduction

Context and motivation In order to be able to cope with process failures, many upper layer services (such as atomic broadcast, atomic commitment, group membership, etc.) rely in one form or another on an underlying basic service called *eventual leader* facility. Such a service provides the processes with a single operation, denoted leader(), such that each invocation of that operation returns a process name, and, after some unknown but finite time, all the invocations returns the same name, and this is the name of an alive process [2].

Building an eventual leader service requires the processes to cooperate in order to elect one of them. It has been shown that such an election is impossible when the progress of each process is totally independent of the progress of the other processes, namely when the processes are fully asynchronous (direct proofs of this impossibility can be found in [1, 16]). So, a fundamental issue consists on finding "as weak as possible" timing assumptions that the underlying system has to satisfy so that an eventual leader service can be built. Answering this question is important from both a practical and theoretical points of view. Seen from a theory point of view, the answer would establish the boundary assumptions beyond which the problem cannot be solved. Seen from a practical point of view, the answer would define the necessary requirements a system has to satisfy in order to solve the problem, and would consequently permit to provide engineers with requirements their systems have to meet.

The message-passing case The previous question has received a lot of attention in the message-passing context, i.e., when the processes cooperate by exchanging messages through an underlying network. Two main approaches have been investigated. One is based on the notion of timely channels, the other is based on the notion of message pattern. The *timely channel* approach (e.g., [1, 3]) requires that, after some time, an appropriate subset of the communication channels behave timely, i.e., they guarantee an upper bound on message transfer delays. The *message pattern* approach (introduced in [14]) requires that some messages are received before other messages. Interestingly, these two approaches can be combined [17] to take the best of "both worlds" and provide eventual leader protocols with an assumption coverage greater than the one provided by any protocol based on a single assumption [18].

The shared memory case The situation is different in the shared memory context. In this case, the only base objects that the processes can use to communicate are atomic read/write registers (also called shared registers). As these registers are intrinsically time-free, the previous question translates as follows: which are the weakest timing assumptions the processes have to satisfy so that an eventual leader service can be built?

To our knowledge, only three eventual leader protocols suited to the shared memory context has been proposed so far [4, 8]. The first protocol (presented in [8]) assumes that there is a finite time after which the processes behave synchronously. So, this timing assumption is pretty strong.

The second paper [4] investigates a much weaker assumption. Basically, it requires that there is eventually a process that behaves synchronously while the other processes can be fully asynchronous provided their timers behave correctly. Two protocols are presented in that paper. In the first protocol, be the execution finite or infinite, all but one shared variables are bounded, and after some time a single process writes forever the shared memory. In the second protocol, all the variables are bounded, but this is obtained at the price of having all the processes to write forever the shared memory. These three previous protocols consider that all the processes (but one) can crash.

Content of the paper Let n denote the total number of processes, and t the maximal number of processes than may crash. Using this notation, the previous protocols implicitly consider $t = n - 1$ (these protocol are said to be wait-free [10], and consequently their code does not use t).

More generally, a protocol is t-resilient if it can cope with up to t faulty processes. (This means that the protocol is guaranteed to work correctly when no more than t process crash. If more processes crash, there is no guarantee and the protocol can behave arbitrarily.) Usually, the system parameter t is explicitly used in the code of a t-resilient protocol. In practice, as the number of processes that crash in a given run is usually very small, it is interesting to design t-resilient protocols.

This paper presents a t-resilient eventual leader protocol ($1 \leq t < n$). When compared to the protocols described in [4], the proposed protocol benefits from the assumption that there are at least $n - t$ correct processes. This additional assumption allows weakening the timing assumption used in [4]. More explicitly, let f be the number of processes that crash in a given run, $0 \leq f \leq t$. While (as the protocols presented in [4]) the proposed protocol requires that the accesses (with respect to a predetermined shared variable) of only one correct process (say p) be eventually synchronous, it requires that the timers of only $t - f$ correct processes (different from p) behave correctly. This is a pretty weak timing assumptions (the protocols presented in [4] require that the timers of all the correct processes behave correctly). A timer behaves incorrectly when it expires arbitrarily, whatever the timeout value it has previously been set to.

This shows that very weak additional assumptions allow implementing an eventual leader facility in a shared memory system. The real-time constraints that have to be satisfied require the synchronous behavior of only one process, and the correct behavior of only $t - f$ timers. As an example, when all the processes allowed to crash do actually crashed (we have then $f = t$), no timer is required to behave correctly (let us notice that, in a very interesting way, this is independent of the value of t).

The protocol enjoys another first class property, namely, it is particularly simple. Moreover, its proof shows very clearly all its possible behaviors, and allows a better understanding of its noteworthy properties.

Roadmap The paper is made up of 6 sections. Section 2 presents the system model and the additional assumption that allows implementing an eventual leader protocol. The protocol is described in Section 3 and proved in Section 4. Section 5 discusses the protocol. Finally, Section 6 concludes the paper.

2 System model, leader election, and additional assumption

2.1 Base asynchronous shared memory model

The system consists of n ($n > 1$) processes denoted $p_1, \ldots, p_n$. The integer i denotes the identity of p_i. A process can fail by *crashing*, i.e., prematurely halting. Until it possibly crashes, a process behaves according to its specification, namely, it executes a sequence of steps as defined by its algorithm. After it has crashed, a process executes no more steps. By definition, a process is *faulty* during a run if it crashes during that run; otherwise it is *correct* in that run. In the following, t denotes the maximum number of processes that are allowed to crash in any run ($1 \leq t \leq n-1$)[1], while f denotes the actual number of processes that crash in a run ($0 \leq f \leq t$).

The processes communicate by reading and writing a memory made up of atomic registers (also called shared variables). Each register is one-writer/multi-reader (1WnR). "1WnR" means that a single process can write into it, but all the processes can read it. (Let us observe that using 1WnR atomic registers is particularly suited for cached-based distributed shared memory.)[2] The only process allowed to write an atomic register is called its owner. *Atomic* means that, although read and write operations on the same register may overlap, each (read or write) operation appears to take effect instantaneously at some point of the time line between its invocation and return events (this is called the *linearization* point of the operation) [11]. Uppercase letters are used for the identifiers of the shared registers. These registers are structured into arrays. As an example, $PROGRESS[i]$ denotes a shared register that can be written only by p_i, and read by any process. A process can have local variables. Those are denoted with lowercase letters, with the process identity appearing as a subscript. As an example, $progress_i$ denotes a local variable of p_i.

Some shared registers are *critical*, while other shared registers are not. A critical register is an atomic register on which some constraint can be imposed by the additional assumptions that allow implementing an eventual leader. This attribute allows restricting the set of registers involved in these assumptions.

This base model is characterized by the fact that there is no assumption on the execution speed of one process with respect to another. This is the classical *asynchronous* crash prone shared memory model. It is denoted $\mathcal{AS}_{n,t}[\emptyset]$ in the following.

2.2 Eventual leader service

The notion of *eventual leader* service has been informally presented in the introduction. It is an entity that provides each process with a primitive leader() that returns a process identity 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* service, denoted Ω, satisfies the following properties [2]. (The second property refers to a notion of global time. It is important to notice that this global time is only for a specification purpose. It is not accessible to the processes.)

- Validity: The value returned by a leader() invocation is a process identity.
- Eventual Leadership: There is a finite time and a correct process p_i such that, after that time, every leader() invocation returns i.
- Termination: Any leader() invocation issued by a correct process terminates.

The Ω leader abstraction has been introduced and formally developed in [2] where it is shown to be the weakest, in terms of information about failures, to solve consensus in asynchronous systems prone to process crashes. Several consensus protocols based on eventual leader service have been proposed (e.g., [7, 12, 15] for message-passing systems, and [5, 9] for shared memory systems).

2.3 Additional behavioral assumption

Underlying intuition As already indicated, Ω cannot be implemented in pure asynchronous systems such as $\mathcal{AS}_{n,t}[\emptyset]$. So, we consider the system is no longer fully asynchronous: its runs satisfy the following assumption denoted EWB (for *e*ventually *w*ell-*b*ehaved). The resulting system is consequently denoted $\mathcal{AS}_{n,t}[EWB]$.[3]

Each process p_i is equipped with a timer denoted $timer_i$. The intuition that underlies EWB is that, on one side, the system has to be "synchronous" enough in order at least one process has a chance to be elected, and, on the complementary side, the others processes must be able to recognize it. The EWB assumption is made up of two parts EWB_1 and EWB_2. EWB_1 is on the existence of a process whose behavior has to satisfy a synchrony property. EWB_2 is on the timers of other processes. EWB_1 and EWB_2 are "matching" properties.

The assumption EWB_1 That assumption restricts the asynchronous behavior of one process. It is defined as follows.

> EWB_1: There are a time τ_{EWB_1}, a bound Δ, and a correct process p_ℓ (τ_{EWB_1}, Δ and p_ℓ may be never explicitly known) such that, after τ_{EWB_1}, any two consecutive write accesses issued by p_ℓ to (its own) critical registers, are completed in at most Δ time units.

[1]This means that, if more than t processes crash in a run, we are outside the system model, and a protocol can then behave arbitrarily. If we want the protocol to cope with any number of process crashes we have to take $t = n-1$.

[2]The atomic registers can also be seen as a high level abstraction of a communication system made up of commodity disks. Such disks can be accessed only by read and write operations. Such "shared memory" systems are described in [6, 13]. Protocols based of commodity disks are described in [5, 9].

[3]Thanks to the knowledge of the system parameter t, this assumption is much weaker than the assumption used in [4].

This property means that, after some arbitrary (but finite) time, the speed of p_ℓ is lower-bounded, i.e., its behavior is partially synchronous (let us notice that, while there is a lower bound, no upper bound is required on the speed of p_ℓ, except the fact that it is not $+\infty$). In the following we say "p_ℓ satisfies EWB_1" to say that p_ℓ is a process that makes satisfied that assumption.

The assumption EWB_2 That assumption, that is on timers, is based on the following timing property. Let a timer be *eventually well-behaved* if there is a time τ_{EWB_2} after which, whatever the duration δ and the time $\tau' \geq \tau_{EWB_2}$ at which it is set to δ, that timer expires at some finite time τ'' such that $\tau'' \geq \tau' + \delta$.

This definition allows a timer to expire at arbitrary times (i.e., times that are unrelated to the duration it has been set to) during an arbitrary but finite time, after which it behaves correctly. We are now in order to state the assumption EWB_2:

> EWB_2: The timers of $(t - f)$ correct processes, different from the process p_ℓ that satisfies EWB_1, are eventually well-behaved.

It is important to see that a well-behaved timer does not impose an upper bound on the duration before which a timer expires (except that it is finite).

When we consider EWB, it is important to notice that any process (but one, that is constrained by a lower bound on its speed) can behave in a fully asynchronous way. Moreover, the local clocks used to implement the timers are not required to be synchronized. Finally, the timers of up to $(n - t) + f$ correct processes can behave arbitrarily. It follows that the timing assumption EWB is particularly weak.

In the following we say "p_x is involved in EWB_2" to say that p_x is a correct process that has an eventually well-behaved timer.

3 A t-resilient eventual leader protocol

3.1 Principle of the protocol

The protocol that implements an eventual leader facility in $\mathcal{AS}_{n,t}[EWB]$ is based on a simple idea: a process p_i elects the process that is the least suspected to have crashed (that idea is used in a lot of eventual leader election protocols in message-passing systems). So, each time a process p_i suspects p_j because it has not observed a progress from p_j during some duration (defined by the latest timeout value used to set its timer), it increases a suspicion counter (denoted $SUSPICIONS[i, j]$).

It is possible that, because its timer does not behave correctly, a process p_i suspects erroneously a process p_k, despite the fact that p_k did some progress (this progress being made visible thanks to assumption EWB_1 if p_k satisfies that assumption). So, when it has to compute its current leader, p_i does not consider all the suspicions. For each process p_k, it takes into account only the $(t + 1)$ smallest values among the n counters $SUSPICIONS[1, k], \ldots, SUSPICIONS[n, k]$. As we will see, due EWB_2, this allows it to eliminate the erroneous suspicions.

As several processes can be equally suspected, p_i uses the function $\mathit{lexmin}(X)$ that outputs the lexicographically smallest pair in the set parameter X, where X is a set of (number of suspicions, process identity) pairs and $(a, i) < (b, j)$ iff $(a < b) \vee (a = b \wedge i < j)$.

3.2 Shared and local variables

Shared variables The shared memory is made up of two arrays of 1WnR shared variables.

- $PROGRESS[1..n]$ is an array of shared variables that contain positive integers. That array is initialized to $[1, \ldots, 1]$. Only p_i can write $PROGRESS[i]$. It does it regularly to indicates to the other processes that it is still alive. Each $PROGRESS[i]$ shared variable is a *critical* variable (see the definition of EWB_1).

- $SUSPICIONS[1..n, 1..n]$ is an array of (non-critical) shared variables that contain positive integers (each entry is initialized to 1). The entries of the vector $SUSPICIONS[i, 1..n]$ can be written only by p_i. $SUSPICIONS[i, j] = x$ means that, up to now, the process p_i has suspected $x - 1$ times the process p_j to have crashed.

Local variables Each process p_i manages the following local variables.

- $progress_i$ is used by p_i to measure its progress, and consequently update $PROGRESS[i]$.

- $last_i[1..n]$ is an array such that $last_i[k]$ contains the latest value of $PROGRESS[k]$ read by p_i.

- $suspicions_i[1..n]$ is an array such that $suspicions_i[k]$ contains the number of times p_i suspected p_k.

- $progress\_k_i$, $timeout_i$ and $susp_i[1..n]$ are auxiliary local variables used by p_i to locally memorize relevant global values.

3.3 Process behavior

The behavior of a process is described in Figure 1. It is decomposed in three tasks. The first task ($T1$) defines the way the current leader is determined. For each process

```
task $T1$:
(1)  when leader() is invoked:
(2)    for_each $k \in \{1, ..., n\}$ do
(3)      let $susp_i[k] = \Sigma$ $(t+1)$ smallest values in the vector $SUSPICIONS[1..n, k]$
(4)    end_for;
(5)    return($\ell$) where $\ell$ is such that $(-, \ell)$ $= lex\_min(\{(susp_i[k], k)\}_{1 \leq k \leq n})$

task $T2$:
(6)  repeat_forever
(7)      $progress_i \leftarrow progress_i + 1$; $PROGRESS[i] \leftarrow progress_i$
(8)  end_repeat

task $T3$:
(9)  when $timer_i$ expires:
(10)   for_each $k \in \{1, ..., n\} \setminus \{i\}$ do
(11)       $progress\_k_i \leftarrow PROGRESS[k]$;
(12)       if $(progress\_k_i \neq last_i[k])$
(13)           then $last_i[k]$                $\leftarrow progress\_k_i$
(14)           else $suspicions_i[k]$          $\leftarrow suspicions_i[k] + 1$;
(15)                $SUSPICIONS[i, k]$         $\leftarrow suspicions_i[k]$
(16)       end_if
(17)   end_for;
(18)   for_each $k \in \{1, ..., n\}$ do
(19)     let $susp_i[k] = \Sigma$ $(t+1)$ smallest values in the vector $SUSPICIONS[1..n, k]$
(20)   end_for;
(21)   let $timeout_i = min(\{susp_i[k]\}_{1 \leq k \leq n})$;
(22)   set $timer_i$ to $timeout_i$
```

Figure 1. A t-resilient eventual leader election algorithm (code for p_i)

p_k, p_i first computes the number of relevant suspicions that concerns p_k. As already mentioned, those are the $(t+1)$ smallest values in the vector $SUSPICIONS[1..n, k]$ (line 3). The current leader is then defined as the process currently the least suspected, when considering only the relevant suspicions (line 5).

The second task ($T2$) is a simple repetitive task whose aim is to increase $PROGRESS[i]$ in order to inform the other processes that p_i has not crashed (line 7). The third task ($T3$) is associated with p_i's timer expiration. It is where p_i possibly suspects the other processes and where it sets its timer ($timer_i$).

1. Suspicion management part (lines 10-17). For each process p_k ($k \neq i$), p_i first reads $PROGRESS[k]$ to see p_k's current progress. If there is no progress since the last reading of $PROGRESS[k]$, p_i suspects once more p_k to have crashed, and increases consequently $SUSPICIONS[i, k]$.

2. Timer setting part (lines 18-22). Then, p_i resets its timer to an appropriate timeout value. That value is computed from the current relevant suspicions. Let us observe that this timeout value increases when these suspicions increase. Let us also remark that, if after some time the number of relevant suspicions does no longer increase, $timeout_i$ keeps forever the same value.

As we can see, the protocol is relatively simple. It uses $n^2 + n$ shared variables. (As, for any i, $SUSPICIONS[i, i]$ is always equal to 1, it is possible to use the diagonal of that matrix to store the array $PROGRESS[1..n]$.)

4 Proof of the protocol

Let us consider a run of the protocol described in Figure 1 in which the assumptions EWB_1 and EWB_2 defined in Section 2.3 are satisfied. This section shows that an eventual leader is elected in that run. The proof is decomposed into several lemmas.

Lemma 1 *Let p_i be a faulty process. For any p_j, $SUSPICIONS[i, j]$ is bounded.*

Proof Let us first observe that the vector $SUSPICIONS[i, 1..n]$ is updated only by p_i. The proof follows immediately from the fact that, after it has crashed, a process does no longer increase shared variables. $\square_{Lemma\ 1}$

Lemma 2 *Let p_i and p_j be a correct and a faulty process, respectively. $SUSPICIONS[i, j]$ grows unboundedly.*

Proof After a process p_j has crashed, it does no longer increase the value of $PROGRESS[j]$, and consequently, due to the update of line 13, there is a finite time after which the test of line 12 remains forever false at any correct process p_i. It follows that $SUSPICIONS[i, j]$ is increased unboundedly at line 15. $\Box_{Lemma\ 2}$

Lemma 3 *Let p_i be a correct process involved in the assumption EWB_2 (i.e., its timer is eventually well-behaved) and let us assume that after some point in time $timer_i$ is always set to a value $\Delta' > \Delta$. Let p_j be a correct process that satisfies the assumption EWB_1. Then, $SUSPICIONS[i, j]$ is bounded.*

Proof As p_i is involved in EWB_2, there is a time τ_{EWB_2} such that $timer_i$ never expires before $\tau + \delta$ if it has been set to δ at time τ, with $\tau \geq \tau_{EWB_2}$. Similarly, as p_j satisfies EWB_1, there are a bound Δ and a time τ_{EWB_1} after which two consecutive write operations issued by p_j on $PROGRESS[j]$ are separated by at most Δ time units (let us recall that $PROGRESS[j]$ is the only critical variable written by p_j).

Let τ_Δ, the time after which $timeout_i$ takes only values $\Delta' > \Delta$, and let $\tau = \max(\tau_\Delta, \tau_{EWB_1}, \tau_{EWB_2})$. As after time τ_Δ any two consecutive write operations into $PROGRESS[j]$ issued by p_j are separated by at most Δ time units, while any two reading of $PROGRESS[j]$ by p_i are separated by at least Δ' time units, it follows that there is a finite a time $\tau' \geq \tau$ after which we always have $PROGRESS[j] \neq last_i[j]$ when evaluated by p_i (line 12). Hence, after τ', the shared variable $SUSPICIONS[i, j]$ is no longer increased, which completes the proof of the lemma. $\Box_{Lemma\ 3}$

Notation 1 Given a process p_k, let $sk_1(\tau) \leq sk_2(\tau) \leq \cdots \leq sk_{t+1}(\tau)$ denote the $(t + 1)$ smallest values among the n values in the vector $SUSPICIONS[1..n, k]$ at time τ. Let $M_k(\tau)$ denote $sk_1(\tau) + sk_2(\tau) + \cdots + sk_{t+1}(\tau)$.

Notation 2 Let S denote the set containing the f faulty processes plus the $(t-f)$ processes involved in the assumption EWB_2 (their timers are eventually well-behaved). Then, for each process $p_k \notin S$, let S_k denote the set $S \cup \{p_k\}$. (Let us notice that $|S_k| = t + 1$.)

Lemma 4 *At any time τ, there is a process $p_i \in S_k$ such that the predicate $SUSPICIONS[i, k] \geq sk_{t+1}(\tau)$ is satisfied.*

Proof Let $K(\tau)$ be the set of the $(t + 1)$ processes p_x such that, at time τ, $SUSPICIONS[x, k] \leq sk_{t+1}(\tau)$. We consider two cases.

1. $S_k = K(\tau)$. Then, taking p_i as the "last" process of S_k such that $SUSPICIONS[i, k] = sk_{t+1}(\tau)$ proves the lemma.

2. $S_k \neq K(\tau)$. in that case, let us take p_i as a process in $S_k \setminus K(\tau)$. As $p_i \notin K(\tau)$, it follows from the definition of $K(\tau)$ that $SUSPICIONS[i, k] \geq sk_{t+1}(\tau)$, and the lemma follows. $\Box_{Lemma\ 4}$

Notation 3 Let $M_x = \max(\{M_x(\tau)_{\tau \geq 0}\})$. If there is no such value ($M_x(\tau)$ grows forever according to τ), let $M_x = +\infty$. Let B be the set of processes p_x such that M_x is bounded.

Lemma 5 *If the assumption EWB_1 is satisfied, then $B \neq \emptyset$.*

Proof Let p_k be a process that satisfies EWB_1. We show that M_k is bounded. Due to Lemma 4, at any time τ, there is a process $p_{j(\tau)} \in S_k$ such that we have $SUSPICIONS[j(\tau), k](\tau) \geq sk_{t+1}(\tau)$. (where $SUSPICIONS[j(\tau), k](\tau)$ denotes the value of the corresponding variable at time τ). It follows that $M_k(\tau)$ is upper bounded by $(t + 1) \times SUSPICIONS[j(\tau), k](\tau)$. So, the proof amounts to show that, after some time, for any $j \in S_k$, $SUSPICIONS[j, k]$ remains bounded. Let us consider any process $p_j \in S_k$ after the time at which the f faulty processes have crashed. There are three cases.

1. $p_j = p_k$. In this case $SUSPICIONS[j, k] = 1$ permanently.

2. p_j is a faulty process of S_k. In that case, the fact that $SUSPICIONS[j, k]$ is bounded follows directly from Lemma 1.

3. p_j is a process of S_k that is one of the $(t - f)$ correct processes involved in the assumption EWB_2. The fact that $SUSPICIONS[j, k]$ is bounded is then an immediate consequence of Lemma 3.

$\Box_{Lemma\ 5}$

Lemma 6 *B does not contain faulty processes.*

Proof Let p_j be a faulty process. Observe that, for each τ, the set of processes whose values in the vector $SUSPICIONS[1..n, j]$ are added to compute $M_j(\tau)$ contains at least one correct process. Due to Lemma 2, for each correct process p_i, $SUSPICIONS[i, j]$ increases forever, and hence so does M_j, which proves the lemma. $\Box_{Lemma\ 6}$

Lemma 7 *There is a time after which any invocation of the primitive* leader() *issued by a correct process, returns the identity of a (correct) process of B.*

Proof The lemma follows from the lines 2-5 and the fact that B is not empty (Lemma 5) and contains only correct processes (Lemma 6). $\Box_{Lemma\ 7}$

Notation 4 Let $(M_a, a) = \textit{lex\_min}(\{(M_x, x) \mid p_x \in B\}$.

Lemma 8 *There is a single process p_a and it is a correct process.*

Proof The lemma follows directly from the following observations: B does not contain faulty processes (Lemma 6), it is not empty (Lemma 5), and no two processes have the same identity. $\Box_{Lemma\ 8}$

Theorem 1 *There is a time after which all the processes elect forever the same correct process as their common leader.*

Proof The theorem follows from Lemma 7 and Lemma 8. $\Box_{Theorem\ 1}$

5 Discussion

On the process that is elected The proof of the protocol relies on the assumption EWB_1 to guarantee that at least one correct process can be elected (i.e., the set B is not empty, -Lemma 5-, and does not contain faulty processes -Lemma 6-). This does not mean that the elected process is a process that satisfies the assumption EWB_1. There are cases where it can be another process.

To see when this can happen, let us consider two correct processes p_i and p_j such that p_i does not satisfy EWB_2 (its timer is never well-behaved) and p_j does not satisfy EWB_1 (it never behaves synchronously). (A re-reading of the statement of Lemma 3 will make the following description easier to understand.) Despite the fact that (1) p_i is not synchronous with respect to a process that satisfies EWB_1, and can consequently suspects these processes infinitely often, and (2) p_j is not synchronous with respect to a process that satisfy EWB_2 (and can consequently be suspected infinitely often by such processes), it is still possible that p_i and p_j behave synchronously one with respect to the other in such a way that p_i never suspects p_j. If this happens $SUSPICIONS[i, j]$ remains bounded, and it is possible that the value M_j not only remains bounded, but becomes the smallest value in the set B. It this occurs, p_j is elected as the common leader.

Of course, there are runs in which the previous scenario does not occur. That is why the protocol has to rely on EWB_1 in order to guarantee that the set B be never empty.

On the timeout values It is important to notice that the timeout values are determined from the least suspected processes. Moreover, after the common leader (say p_y) has been elected, any timeout value is set to M_y. It follows that, be the run finite or infinite, the timeout values are always bounded.

6 Conclusion

This paper was on the election of an eventual leader in asynchronous systems in which the processes communicate by reading and writing atomic registers only, and where up to t processes can crash ($1 \leq t < n$, n being the total number of processes). As this problem cannot be solved in pure (message-passing or shared memory) asynchronous systems, the paper has proposed and investigated an additional timing assumption that allows electing an eventual leader at least in the asynchronous runs that satisfy it. The assumption requires that after some time (i) there is process that behaves synchronously, and (ii) $(t - f)$ other processes have timers that work correctly (where f is the actual number of process crashes in a run). A timer works incorrectly when it expires too with respect to the value it has been set. A t-resilient protocol that benefits from this assumption has been proposed and proved correct. A noteworthy property of this protocol lies in its design simplicity.

Among problems that remain open, we list two. One concerns the values taken by the suspicion-related shared variables ($SUSPICIONS[i, j]$): it is possible to bound their domain? If the answer is "yes", how can it be done? Another problem would consist in finding a common framework that would unify the protocol presented in this paper and the protocols introduced in [4].

References

[1] 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, 2004.

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

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

[4] Fernández A., Jiménez E. and Raynal M., Electing an Eventual Leader in an Asynchronous Shared Memory System. *Tech Report #1821*, 19 pages, IRISA, University of Rennes 1, October 2006.

[5] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20, 2003.

[6] Gibson G.A. *et al.*, A Cost-effective High-bandwidth Storage Architecture. *Proc. 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, ACM Press, pp. 92-103, 1998.

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

[8] Guerraoui R. and Raynal M., A Leader Election Protocol for Eventually Synchronous Shared Memory Systems. *4th Int'l IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'06)*, IEEE Computer Society Press, pp. 75-80, 2006.

[9] Guerraoui R. and Raynal M., The Alpha of Asynchronous Consensus. *The Computer Journal*, To appear, 2007.

[10] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on programming Languages and Systems*, 11(1):124-149, 1991.

[11] Herlihy M.P. and Wing J.M, Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[12] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998. (The first version of Paxos appeared a a DEC Tech Report in 1989.)

[13] Lee E.K. and Thekkath C., Petal: Distributed Virtual Disks. *Proc. 7th Int'l Conference on Architectural Support for Programing Languages and Operating Systems (ASPLOS'96)*, ACM Press, pp. 84-92, 1996.

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

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

[16] Mostéfaoui A., Raynal M. and Travers C., Crash Resilient Time-Free Eventual Leadership. *Proc. 23th Symposium on Resilient Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 208-218, 2004.

[17] Mostéfaoui A., Raynal M. and Travers C., Time-free and Timeliness Assumptions can be Combined to Get Eventual Leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656-666, 2006.

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