Electing an Eventual Leader in an Asynchronous Shared Memory System *

Antonio FERNÁNDEZ† Ernesto JIMÉNEZ‡ Michel RAYNAL⋆

† 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

Abstract

This paper considers the problem of electing an eventual leader in an asynchronous shared memory system. While this problem has received a lot of attention in message-passing systems, very few solutions have been proposed for shared memory systems. As an eventual leader cannot be elected in a pure asynchronous system prone to process crashes, the paper first proposes to enrich the asynchronous system model with an additional assumption. That assumption, denoted AWB, requires that after some time (1) there is a process whose write accesses to some shared variables are timely, and (2) the timers of the other processes are asymptotically well-behaved. The asymptotically well-behaved timer notion is a new notion that generalizes and weakens the traditional notion of timers whose durations are required to monotonically increase when the values they are set to increase. Then, the paper presents two AWB-based algorithms that elect an eventual leader. Both algorithms are independent of the value of t (the maximal number of processes that may crash). The first algorithm enjoys the following noteworthy properties: after some time only the elected leader has to write the shared memory, and all but one shared variables have a bounded domain, be the execution finite or infinite. This algorithm is consequently optimal with respect to the number of processes that have to write the shared memory. The second algorithm enjoys the following property: all the shared variables have a bounded domain. This is obtained at the following additional price: all the processes are required to forever write the shared memory. A theorem is proved which states that this price has to be paid by any algorithm that elects an eventual leader in a bounded shared memory model. This second algorithm is consequently optimal with respect to the number of processes that have to write in such a constrained memory model. In a very interesting way, these algorithms show an inherent tradeoff relating the number of processes that have to write the shared memory and the bounded/unbounded attribute of that memory.

1 Introduction

Equipping an asynchronous system with an oracle An asynchronous system is characterized by the absence of a bound on the time it takes for a process to proceed from a step of its algorithm to the next one. Combined with process failures, such an absence of a bound can make some synchronization or coordination problems impossible to solve (even when the processes communicate through a reliable communication medium). The most famous of these "impossible" asynchronous problems is the well-known *consensus* problem [7]. Intuitively, this impossibility comes from the fact that a process cannot safely distinguish a crashed process from a very slow process.

One way to address and circumvent these impossibilities consists on enriching the underlying asynchronous systems with an appropriate *oracle* [27]. More precisely, in a system prone to process failures, such an oracle (sometimes called *failure detector*) provides each process with hints on which processes are (or are not) faulty. According to the quality of these hints, several classes of oracles can be defined [3]. So, given an asynchronous system prone to process failures equipped with an appropriate oracle, it becomes possible to solve a problem that is, otherwise, impossible to solve in a purely asynchronous system. This means that an oracle provides processes with additional computability power.

Fundamental issues related to oracles for asynchronous systems Two fundamental questions can be associated

*The work of A. Fernández and E. Jiménez was partially supported by the Spanish MEC under grants TIN2005-09198-C02-01, TIN2004-07474-C02-02, and TIN2004-07474-C02-01, and the Comunidad de Madrid under grant S-0505/TIC/0285. The work of Michel Raynal was supported by the European Network of Excellence ReSIST.

†The work of this author was done while on leave at IRISA, supported by the Spanish MEC under grant PR-2006-0193.

with oracles. The first is more on the theoretical side and concerns their computability power. Given a problem (or a family of related problems), which is the weakest oracle that allows solving that problem in an asynchronous system where processes can experience a given type of failures? Intuitively, an oracle O_w is the weakest for solving a problem P if it allows solving that problem, and any other oracle O_{nw} that allows solving P provides hints on failures that are at least as accurate as the ones provided by O_w (this means that the properties defining O_{nw} imply the ones defining O_w, but not necessarily vice-versa). It has been shown that, in asynchronous systems prone to process crash failures, the class of *eventual leader* oracles is the weakest for solving asynchronous *consensus*, be these systems message-passing systems [4] or shared memory systems [20][1]. It has also been shown that, for the same type of process failures, the class of *perfect failure detectors* (defined in [3]) is the weakest for solving asynchronous *interactive consistency* [14].

The second important question is on the algorithm/protocol side and concerns the implementation of oracles (failure detectors) that are designed to equip an asynchronous system. Let us first observe that no such oracle can be implemented on top of a purely asynchronous system (otherwise the problem it allows solving could be solved in a purely asynchronous system without additional computability power). So, this fundamental question translates as follows. First, find "reasonably weak" behavioral assumptions that, when satisfied by the underlying asynchronous system, allow implementing the oracle. "Reasonably weak" means that, although they cannot be satisfied by all the runs, the assumptions are actually satisfied in "nearly all" the runs of the asynchronous system. Second, once such assumptions have been stated, design efficient algorithms that implement correctly the oracle in all the runs satisfying the assumptions.

Content of the paper Considering the asynchronous shared memory model where any number of processes can crash, this paper addresses the construction of eventual leader oracles [4]. Such an oracle (usually denoted Ω)[2] provides the processes with a primitive $\mathsf{leader}()$ that returns a process identity, and satisfies the following "eventual" property in each run R: There is a time after which all the invocations of $\mathsf{leader}()$ return the same identity, that is the identity of a process that does not crash in the run R.

As already indicated, such an oracle is the weakest to solve the consensus problem in an asynchronous system where processes communicate through single-writer/multi-readers (1WnR) atomic registers and are prone to crash failures [20].

The paper has three main contributions.

- It first proposes a behavioral assumption that is particularly weak. This assumption is the following one. In each run, there are a finite (but unknown) time τ and a process p (not a priori known) that does not crash in that run, such that after τ:
  - (1) There is a bound Δ (not necessarily known) such that any two consecutive write accesses to some shared variables issued by p are separated by at most Δ time units, and
  - (2) Each correct process $q \neq p$ has a timer that is *asymptotically well-behaved*. Intuitively, this notion expresses the fact that eventually the duration that elapses before a timer expires has to increase when the timeout parameter increases.

  It is important to see that the timers can behave arbitrarily during arbitrarily long (but finite) periods. Moreover, as we will see in the formal definition, their durations are not required to strictly increase according to their timeout periods. After some time, they have only to be lower-bounded by some monotonously increasing function.

  It is noteworthy to notice that no process (but p) is required to have any synchronous behavior. Only their timers have to eventually satisfy some (weak) behavioral property.

- The paper then presents two algorithms that construct an Ω oracle in all the runs that satisfy the previous behavioral assumptions, and associated lower bounds. All the algorithms use atomic 1WnR atomic registers. The algorithms, that are of increasing difficulty, are presented incrementally.
  - In the first algorithm, all (but one of) the shared variables have a bounded domain (the size of which depends on the run). More specifically, this means that, be the execution finite or infinite, even the timeout values stop increasing forever.

    Moreover, after some time, there is a single process that writes the shared memory. The algorithm is consequently write-efficient. It is even write-optimal as at least one process has to write the shared memory to inform the other processes that the current leader is still alive.
  - The second algorithm improves the first one in the sense that all the (local and shared) variables are bounded. This nice property is obtained by

[1] Let us also notice that the Paxos fault-tolerant state machine replication algorithm [18] is based on the Ω abstraction. For the interested reader, an introduction to the family of Paxos algorithms can be found in [12].

[2] Without ambiguity and according to the context, Ω is used to denote either the class of eventual leader oracles, or an oracle of that class.

using two boolean flags for each pair of processes. These flags allow each process p to inform each other process q that it has read some value written by q.

- The third contribution is made up of lower bound results are proved for the considered model. Two theorems are proved that state (1) the process that is eventually elected has to forever write the shared memory, and (2) any process (but the eventual leader) has to forever read from the shared memory. Another theorem shows that, if the shared memory is bounded, then all the processes have to forever write into the shared memory. These theorems show that both the algorithms presented in the paper are optimal with respect to these criteria.

Why shared memory-based Ω algorithms are important Multi-core architectures are becoming more and more deployed and create a renewed interest for asynchronous shared memory systems. In such a context, it has been shown [10] that Ω constitutes the weakest *contention manager* that allows transforming any obstruction-free [15] software transactional memory into a non-blocking transactional memory [16]. This constitutes a very strong motivation to look for requirements that, while being "as weak as possible", are strong enough to allow implementing Ω in asynchronous shared memory environments prone to process failures.

On another side, some distributed systems are made up of computers that communicate through a network of attached disks. These disks constitute a storage area network (SAN) that implements a shared memory abstraction. As commodity disks are cheaper than computers, such architectures are becoming more and more attractive for achieving fault-tolerance. The Ω algorithms presented in this paper are suited to such systems [9].

Related work As far as we know, a single shared memory Ω algorithm has been proposed so far [13]. This algorithm considers that the underlying system satisfies the following behavioral assumption: there is a time τ after which there are a lower bound and an upper bound for any process to execute a local step, or a shared memory access. This assumption defines an eventually synchronous shared memory system. It is easy to see that it is a stronger assumption than the assumption previously defined here.

The implementation of Ω in asynchronous message-passing systems is an active research area. Two main approaches have been been investigated: the *timer*-based approach and the *message pattern*-based approach.

The timer-based approach relies on the addition of timing assumptions [5]. Basically, it assumes that 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. The algorithms implementing Ω in such "augmented" asynchronous systems are based on timeouts (e.g., [1, 19]). They use successive approximations to eventually provide each process with an upper bound on transfer delays and processing speed. They differ mainly on the "quantity" of additional synchrony they consider, and on the message cost they require after a leader has been elected.

Among the protocols based on this approach, a protocol presented in [1] is particularly attractive, as it considers a relatively weak additional synchrony requirement. Let t be an upper bound on the number of processes that may crash ($1 \leq t < n$, where n is the total number of processes). This assumption is the following: the underlying asynchronous system, which can have fair lossy channels, is required to have a correct process p that is a $\diamond t$-*source*. This means that p has t output channels that are eventually timely: there is a time after which the transfer delays of all the messages sent on such a channel are bounded (let us notice that this is trivially satisfied if the receiver has crashed). Notice that such a $\diamond t$-*source* is not known in advance and may never be explicitly known. It is also shown in [1] that there is no leader protocol if the system has only $\diamond(t-1)$-*source*s. A versatile adaptive timer-based approach has been developed in [21].

The message pattern-based approach, introduced in [22], does not assume eventual bounds on process and communication delays. It considers that there is a correct process p and a set Q of t processes (with $p \notin Q$, moreover Q can contain crashed processes) such that, each time a process $q \in Q$ broadcasts a query, it receives a response from p among the first $(n-t)$ corresponding responses (such a response is called a winning response). It is easy to see that this assumption does not prevent message delays to always increase without bound. Hence, it is incomparable with the synchrony-related $\diamond t$-*source* assumption. This approach has been applied to the construction of an Ω algorithm in [24].

A *hybrid* algorithm that combines both types of assumption is developed in [25]. More precisely, this algorithm considers that each channel eventually is timely or satisfies the message pattern, without knowing in advance which assumption it will satisfy during a particular run. The aim of this approach is to increase the assumption coverage, thereby improving fault-tolerance [26].

Roadmap The paper is made up of 5 sections. Section 2 presents the system model and the additional behavioral assumption. Then, Sections 3 and 4 present in an incremental way the two algorithms implementing an Ω oracle, and show they are optimal with respect to the number of processes that have to write or read the shared memory. Finally,

Section 5 provides concluding remarks.

Due to page limitation, the proofs of some lemmas and theorems are omitted. The reader can find them in [6].

2 Base Model, Eventual Leader and Additional Behavioral 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. (Sometimes a process is also denoted p, q or r.) 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. There is no assumption on the maximum number t of processes that may crash, which means that up to $n-1$ process may crash in a run.

The processes communicate by reading and writing a memory made up of atomic registers (also called shared variables in the following). 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.) 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) [17]. 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.

Some shared registers are *critical*, while other shared registers are not. A critical register is a 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.

A process can have local variables. They are denoted with lowercase letters, with the process identity appearing as a subscript. As an example, $candidates_i$ denotes a local variable of p_i.

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 $\mathcal{AS}_n[\emptyset]$ in the following.

2.2 Eventual leader service

The notion of *eventual leader* oracle has been informally presented in the introduction. It is an entity that provides each process with a primitive $\mathsf{leader}()$ 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* oracle, denoted Ω, satisfies the following property [4]:

- Validity: The value returned by a $\mathsf{leader}()$ invocation is a process identity.
- Eventual Leadership[3]: There is a finite time and a correct process p_i such that, after that time, every $\mathsf{leader}()$ invocation returns i.
- Termination: Any $\mathsf{leader}()$ invocation issued by a correct process terminates.

The Ω leader abstraction has been introduced and formally developed in [4] where it is shown to be the weakest, in terms of information about failures, to solve consensus in asynchronous systems prone to process crashes (assuming a majority of correct processes). Several Ω-based consensus protocols have been proposed (e.g., [11, 18, 23] for message-passing systems, and [8] for shared memory systems)[4].

2.3 Additional behavioral assumption

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

Each process p_i is equipped with a timer denoted $timer_i$. The intuition that underlies AWB is that, once a process p_ℓ is defined as being the current leader, it should not to be demoted by a process p_i that believes p_ℓ has crashed. To that end, constraints have to be defined on the behavior of both p_ℓ and p_i. The constraint on p_ℓ is to force it to "regularly" inform the other processes that it is still alive. The constraint on a process p_i is to prevent it to falsely suspect that p_ℓ has crashed.

There are several ways to define runs satisfying the previous constraints. As an example, restricting the runs to

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

[4] It is important to notice that, albeit it can be rewritten using Ω (first introduced in 1992), the original version of Paxos, that dates back to 1989, was not explicitly defined with this formalism. The first paper where Paxos is explained as an Ω-based algorithm is [2].

be "eventually synchronous" would work but is much more constraining than what is necessary. The aim of the AWB additional assumption is to state constraints that are "as weak as possible"[5]. It appears that requiring the timers to be eventually monotonous is stronger than necessary (as we are about to see, this is a particular case of the AWB assumption). The AWB assumption is made up of two parts AWB_1 and AWB_2 that we present now. AWB_1 is on the existence of a process whose behavior has to satisfy a synchrony property. AWB_2 is on the timers of the other processes. AWB_1 and AWB_2 are "matching" properties.

The assumption AWB_1 The AWB_1 assumption requires that eventually a process does not behave in a fully asynchronous way. It is defined as follows.

> AWB_1: There are a time τ_{01}, a bound Δ, and a correct process p_ℓ (τ_{01}, Δ and p_ℓ may be never explicitly known) such that, after τ_{01}, any two consecutive write accesses issued by p_ℓ to (its own) critical registers, are completed in at most Δ time units.

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 $+\infty$).

The assumption AWB_2 In order to define AWB_2, we first introduce a function $f()$ with monotonicity properties that will be used to define an asymptotic behavior. That function takes two parameters, a time τ and a duration x, and returns a duration. It is defined as follows. There are two (possibly unknown) bounded values x_f and τ_f such that:

- (f1) $\forall \tau_2, \tau_1 : \tau_2 \geq \tau_1 \geq \tau_f, \forall x_2, x_1 : x_2 \geq x_1 \geq x_f$: $f(\tau_2, x_2) \geq f(\tau_1, x_1)$. (After some point, $f()$ is not decreasing with respect to τ and x).
- (f2) $\lim_{x \to +\infty} f(\tau_f, x) = +\infty$. (Eventually, $f()$ always increases[6].)

We are now in order to define the notion of *asymptotically well-behaved* timer. Considering the timer $timer_i$ of a process p_i and a run R, let τ be a real time at which the timer is set to a value x, and τ' be the finite real time at which that timer expires. Let $T_R(\tau, x) = \tau' - \tau$, for each x and τ. Then timer $timer_i$ is asymptotically well-behaved in a run R, if there is a function $f_R()$, as defined above, such that:

- (f3) $\forall \tau : \tau \geq \tau_f, \forall x : x \geq x_f$: $f_R(\tau, x) \leq T_R(\tau, x)$.

This constraint states the fact that, after some point, the function $T_R()$ is always above the function $f_R()$. It is important to observe that, after (τ_f, x_f), the function $T_R(\tau, x)$ is not required to be non-decreasing, it can increase and decrease. Its only requirement is to always dominate $f_R()$. (See Figure 1.)

> AWB_2: The timer of each correct process (except possibly p_ℓ) is asymptotically well-behaved.

When we consider AWB, it is important to notice that any process (but p_ℓ constrained by a speed lower bound) can behave in a fully asynchronous way. Moreover, the local clocks used to implement the timers are required to be neither synchronized, nor accurate with respect to real-time.

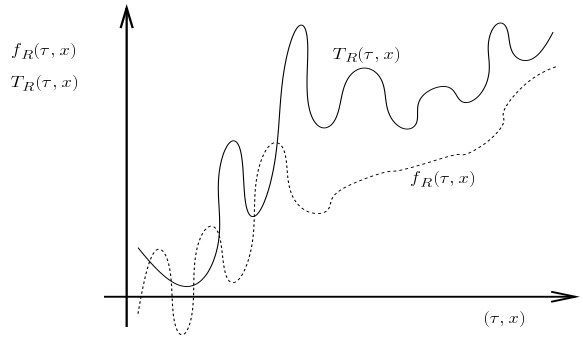

Figure 1. $T_R()$ **asymptotically dominates** $f_R()$

3 An Ω algorithm for $\mathcal{AS}_n[AWB]$

3.1 Principles of the algorithm

The first algorithm implementing Ω in $\mathcal{AS}_n[AWB]$ that we present, relies on a very simple idea that has been used in several algorithms that build Ω in message-passing systems. Each process p_i handles a set ($candidates_i$) containing the processes that (from its point of view) are candidates for being the leader. When it suspects one of its candidates p_j to have crashed, p_i makes public the fact that it suspects p_j once more. (This is done by p_i increasing the shared register $SUSPICIONS[i, j]$.)

Finally, a process p_i defines its current leader as the least suspected process among its current candidates. As several processes can be equally suspected, p_i uses the function *lexmin*(X) that outputs the lexicographically smallest pair in the set parameter X, where X is the set of (number of suspicions, process identity) pairs defined from $candidate_i$, and $(a, i) < (b, j)$ iff $(a < b) \vee (a = b \wedge i < j)$.

[5] Of course, the notion of "as weak as possible" has to be taken with its intuitive meaning. This means that, when we want to implement Ω in a shared memory system, we know neither an assumption weaker than AWB, nor the answer to the question: Is AWB the weakest additional assumption?

[6] If the image of $f()$ is the set of natural numbers, then this condition can be replaced by $x_2 > x_1 \implies f(\tau_f, x_2) > f(\tau_f, x_1)$.

3.2 Description of the algorithm

The algorithm, based on the principles described just above, that builds Ω in $\mathcal{AS}_n[AWB]$ is depicted in Figure 2.

Shared variables The variables shared by the processes are the following:

- $SUSPICIONS[1..n, 1..n]$ is an array of natural registers. $SUSPICIONS[j,k] = x$ means that, up to now, p_j has suspected x times the process p_k to have crashed. The entries $SUSPICIONS[j,k]$, $1 \leq k \leq n$ can be written only by p_j.
- $PROGRESS[1..n]$ is an array of natural registers. Only p_i can write $PROGRESS[i]$. (It does it only when it considers it is the leader.)
- $STOP[1..n]$ is an array of boolean registers. Only p_i can write $STOP[i]$. It sets it to $false$ to indicate it considers itself as leader, and sets it to $true$ to indicate it stops considering it is the leader.

The initial values of the previous shared variables could be arbitrary[7]. To improve efficiency, we consider that the natural integer variables are initialized to 1 and the boolean variables to $true$.

Each shared register $PROGRESS[k]$ or $STOP[k]$, $1 \leq k \leq n$ is critical. Differently, none of the registers $SUSPICIONS[j,k]$, $1 \leq j,k \leq n$, is critical. This means that, for a process p_k involved in the assumption AWB_1, only the write accesses to its registers $PROGRESS[k]$ and $STOP[k]$ are concerned.

Let us observe that, as the shared variables $PROGRESS[i]$, $STOP[i]$ and $SUSPICIONS[i,k]$, $1 \leq k \leq n$, are written only by p_i, that process can save their values in local memory and, when it has to read any of them, it can read instead its local copy. (We do not do it in our description of the algorithms to keep simpler the presentation.)

Process behavior The algorithm is made up of three tasks. Each local variable $candidate_i$ is initialized to any set of process identities containing i.

The task $T1$ implements the leader() primitive. As indicated, p_i determines the least suspected among the processes it considers as candidates (lines 02-04), and returns its identity (line 05).

[7]This means that the algorithm is *self-stabilizing* with respect to the *shared* variables. Whatever their initial values, it converges in a finite number of steps towards a common leader, as soon as the additional assumption is satisfied. When these variables have arbitrary initial values (that can be negative), line 27 of Figure 2 has to be "set $timer_i$ to $\max(1, \max\{SUSPICIONS[i,k]\}_{1\leq k\leq n})$" in order a timer be never set to a negative value.

The task $T2$ is an infinite loop. When it considers it is the leader, (line 07), p_i repeatedly increases $PROGRESS[i]$ to inform the other processes that it is still alive (lines 07-10). If it discovers it is no longer leader, p_i sets $STOP[i]$ to $true$ (line 11) to inform the other processes it is no longer competing to be leader.

```
task T1:
(01) when leader() is invoked:
(02)   for_each k ∈ candidates_i do
(03)       susp_i[k] ← Σ_{1≤j≤n} SUSPICIONS[j,k] end_for;
(04)   let (−, ℓ) = lex_min({(susp_i[k], k)}_{k∈candidates_i});
(05)   return(ℓ)

task T2:
(06) repeat_forever
(07)   while (leader() = i) do
(08)       PROGRESS[i] ← PROGRESS[i] + 1;
(09)       if STOP[i] then STOP[i] ← false end_if
(10)   end_while;
(11)   if (¬ STOP[i]) then STOP[i] ← true end_if
(12) end_repeat

task T3:
(13) when timer_i expires:
(14)   for_each k ∈ {1, ..., n} \ {i} do
(15)     stop_k_i     ← STOP[k];
(16)     progress_k_i ← PROGRESS[k];
(17)     if (progress_k_i ≠ last_i[k]) then
(18)       candidates_i ← candidates_i ∪ {k};
(19)       last_i[k]    ← progress_k_i
(20)     else_if (stop_k_i) then
(21)       candidates_i ← candidates_i \ {k}
(22)     else_if (k ∈ candidates_i) then
(23)       SUSPICIONS[i,k] ← SUSPICIONS[i,k] + 1;
(24)       candidates_i    ← candidates_i \ {k}
(25)     end_if
(26)   end_for;
(27)   set timer_i to max({SUSPICIONS[i,k]}_{1≤k≤n})
```

Figure 2. Write-efficient, all variables are 1WMR, bounded except a single entry of the shared array $PROGRESS[1..n]$ (code for p_i)

Each process p_i has a local timer (denoted $timer_i$), and manages a local variable $last_i[k]$ where it saves the greatest value that it has ever read from $PROGRESS[k]$. The task $T3$ is executed each time that timer expires (line 13). Then, p_i executes the following statements with respect to each process p_k (but itself, see line 14). First, p_i checks if p_k did some progress since the previous timer expiration (line 17). Then, it does the following.

- If $PROGRESS[k]$ has progressed, p_i considers p_k as a candidate to be leader. To that end it adds k to the local set $candidates_i$ (line 18). (It also updates $last_i[k]$, line 19.)
- If $PROGRESS[k]$ has not progressed, p_i checks the

value of $STOP[k]$ (line 20). If it is true, p_k voluntarily demoted itself from being a candidate. Consequently, p_i suppresses k from its local set $candidates_i$ (line 21). If $STOP[k]$ is false and p_k is candidate from p_i's point of view (line 22), p_i suspects p_k to have crashed (line 23) and suppresses it from $candidates_i$ (line 24).

Then, p_i resets its local timer (line 27). Let us observe that no variable of the array $SUSPICIONS$ can decrease and such an entry is increased each time a process is suspected by another process. Thanks to these properties, we will see in the proof that $\max(\{SUSPICIONS[i,k]\}_{1\leq k\leq n})$ can be used as the next timeout value. Note that to compute this value only variables owned by p_i are accessed.

3.3 Proof of the algorithm

Lemma 1 [6] *Let p_k be a faulty process and p_i a correct process. Eventually, the predicate $k \notin candidates_i$ remains true forever.*

Given a run R and a process p_x, let M_x denote the largest value ever taken by $\Sigma_{1\leq j\leq n} SUSPICIONS[j,x]$. If there is no such value (i.e., $\Sigma_{1\leq j\leq n} SUSPICIONS[j,x]$ grows forever), let $M_x = +\infty$. Finally, let B be the set of correct processes p_x such that $M_x \neq +\infty$ (B stands for "bounded").

Lemma 2 [6] *Let us assume that the behavioral assumption AWB is satisfied. Let p_i be a process that satisfies assumption AWB_1. Then, $i \in B$ and, hence, $B \neq \emptyset$.*

Let $(M_\ell, \ell) = lexmin(\{M_x, x) \mid x \in B\})$.

Lemma 3 [6] *There is a single process p_ℓ and it is correct.*

Lemma 4 [6] *There is a time after which p_ℓ permanently executes the loop defined by the lines 07-10 of task $T2$.*

Theorem 1 *There is a time after which a correct process is elected as the eventual common leader.*

Proof We show that p_ℓ is the eventual common leader. From Lemma 3 p_ℓ is unique and correct. Moreover, due the definitions of the bound M_ℓ and the set B, there is a finite time τ after which, for each correct process p_i, $i \neq \ell$, we have $\big(\Sigma_{1\leq j\leq n} SUSPICIONS[j,i], i\big) > (M_\ell, \ell)$. Moreover, due to Lemma 1, there is a time after which, for each correct process p_i and each faulty process p_k we have $k \notin candidate_i$. It follows from these observations, that proving the theorem amounts to show that eventually the predicate $\ell \in candidate_i$ remains permanently true at each correct process p_i.

Let us notice that the predicate $x \in candidate_x$ is always true for any process p_x. This follows from the fact that initially x belongs to $candidate_x$, and then p_x does not execute the tasks $T3$ for $k = x$, and consequently cannot withdraw x from $candidate_x$. It follows that we always have $\ell \in candidate_\ell$. So, let us examine the case $i \neq \ell$.

It follows from Lemma 4 that there is a time τ after which p_ℓ remains permanently in the **while** loop of task $T2$. Let $\tau' \geq \tau$ be a time at which we have $\Sigma_{1\leq j\leq n} SUSPICIONS[j,\ell] = M_\ell$, and p_ℓ has executed line 09 (i.e., $STOP[\ell]$ remains false forever).

After τ', because p_ℓ is forever increasing $PROGRESS[\ell]$, the test of line 17 eventually evaluates to true and (if not already done) p_i adds ℓ to $candidate_i$. We claim that, after that time, the task $T3$ of p_i is always executing the lines 18-19 (for $k = \ell$), from which it follows that ℓ remains forever in $candidate_i$.

Proof of the claim. Let us assume by contradiction that the test of line 17 is false when evaluated by p_i. It follows that ℓ is withdrawn from $candidate_i$, and this occurs at line 24. (It cannot occur at line 21 because after τ we always have $STOP[\ell] = false$.) But line 23 is executed before 24, from which we conclude that $SUSPICIONS[i,\ell]$ has been increased, which means that we have now $\Sigma_{1\leq j\leq n} SUSPICIONS[j,\ell] = M_\ell + 1$, contradicting the definition of the bound M_ℓ. *End of the proof of the claim.* $\square_{Theorem\ 1}$

Theorem 2 [6] *Let p_ℓ be the eventual common leader. All shared variables (but $PROGRESS[\ell]$) are bounded.*

Theorem 3 [6] *After a finite time, only one process (the eventual common leader) writes forever into the shared memory. Moreover, it always writes the same shared variable.*

3.4 Optimality Results

Let $\mathcal{A}$ be any algorithm that implements Ω in $\mathcal{AS}_n[AWB]$ with up to t faulty processes. We have the following lower bounds.

Lemma 5 *Let R be any run of $\mathcal{A}$ with less than t faulty processes and let p_ℓ be the leader chosen in R. Then p_ℓ must write forever in the shared memory in R.*

Lemma 6 [6] *Let R be any run of $\mathcal{A}$ with less than t faulty processes and let p_ℓ be the leader chosen in R. Then every correct process p_i, $i \neq \ell$, must read forever from the shared memory in R.*

The following theorem follows immediately from the previous lemmas.

Theorem 4 [6] *The algorithm described in Figure 2 is optimal in with respect to the number of processes that have to write the shared memory. It is quasi-optimal with respect to the number of processes that have to read the shared memory.*

The "quasi-optimality" comes from the fact that the algorithm described in Figure 2 requires that each process (including the leader) reads forever the shared memory (all the processes have to read the array $SUSPICIONS[1..n, 1..n]$).

3.5 Discussion

Using multi-writer/multi-reader (nWnR) atomic registers If we allow nWnR atomic variables, each column $SUSPICIONS[*, j]$ can be replaced by a single $SUSPICIONS[j]$. Consequently vectors of nWnR atomic variables can be used instead of matrices of 1WnR atomic variables.

Eliminating the local clocks The timers (and consequently the local clocks used to implement them) can be eliminated as follows. Each $timer_i$ is now a local variable managed by p_i as follows (where each execution of the statement $timer_i \leftarrow timer_i - 1$ is assumed to take at least one time unit). The code of task $T3$ becomes accordingly:

task $T3$: $timer_i \leftarrow 1$;
while $(true)$ **do**
  $timer_i \leftarrow timer_i - 1$;
  if $(timer_i = 0)$
    then Line 14 until Line 26 of Figure 2 or 3;
      $timer_i \leftarrow \max(\{SUSPICIONS[i, k]\}_{1 \leq k \leq n})$
  end_if
end_while.

4 An Ω algorithm for $\mathcal{AS}_n[AWB]$ with Bounded Variables Only

4.1 A Lower Bound Result

This section shows that any algorithm that implements Ω in $\mathcal{AS}_n[AWB]$ with only bounded memory requires all correct processes to read and write the shared memory forever. As we will see, it follows from this lower bound that the algorithm described in Figure 3 is optimal with respect to this criterium.

Let $\mathcal{A}$ be an algorithm that implements Ω in $\mathcal{AS}_n[AWB]$ such that, in every run R of $\mathcal{A}$, the number of shared memory bits used is bounded by a value S_R (which may depend on the run). This means that in any run there is time after which no new memory positions are used, and each memory position has bounded number of bits. To make the result stronger, we also assume that $\mathcal{A}$ knows t (maximum number of processes that can fail in any run of $\mathcal{A}$).

Theorem 5 [6] *The algorithm $\mathcal{A}$ has runs in which at least $t + 1$ processes write forever in the shared memory.*

The system model defined in this paper assumes $t = n - 1$. Hence the following corollary.

Corollary 1 *Any algorithm that implements Ω in $\mathcal{AS}_n[AWB]$ with bounded shared memory has runs in which all processes write the shared memory forever.*

4.2 An algorithm with only bounded variables

Principles and description As already indicated, we are interested here in an algorithm whose variables are all bounded. To attain this goal, we use a hand-shaking mechanism. More precisely, we replace the shared array $PROGRESS[1..n]$ and all the local arrays $last_i[1..n]$, $1 \leq i \leq n$, by two shared matrices of 1WnR boolean values, denoted $PROGRESS[1..n, 1..n]$ and $LAST[1..n, 1..n]$.

The hand-shaking mechanism works a follows. Given a pair of processes p_i and p_k, $PROGRESS[i, k]$ and $LAST[i, k]$ are used by these processes to send signals to each other. More precisely, to signal p_k that it is alive, p_i sets $PROGRESS[i, k]$ equal to $\neg LAST[i, k]$. In the other direction, p_k indicates that it has seen this "signal" by cancelling it, namely, it resets $LAST[i, k]$ equal to $PROGRESS[i, k]$. It follows from the essence of the hand-shaking mechanism that both p_i and p_k have to write shared variables, but as shown by Corollary 1, this is the price that has to be paid to have bounded shared variables.

Using this simple technique, we obtain the algorithm described in Figure 3. In order to capture easily the parts that are new or modified with respect to the previous algorithm, the line number of the new statements are suffixed with the letter R (so the line 08 of the previous protocol is replaced by three new lines, while each of the lines 16, 17 and 19 is replaced by a single line). This allows a better understanding of the common principles on which both algorithms rely.

Proof of the algorithm The statement of the lemmas 1, 2, 3 and 4, and Theorem 1 are still valid when the shared array $PROGRESS[1..n]$ and the local arrays $last_i[1..n]$, $1 \leq i \leq n$ are replaced by the shared matrices $PROGRESS[1..n, 1..n]$ and $LAST[1..n, 1..n]$.

As far as their proofs are concerned, the proofs of the lemmas 3 and 4 given in Section 3.3 are verbatim the same. The proofs of the lemmas 1 and 2, and the proof of Theorem 1 have to be slightly modified to suit to the new context. Basically, they differ from their counterparts of Section 3.3 in the way they establish the property that, after some time, no correct process p_i misses an "alive" signal from a process that satisfies the assumption AWB_1. (More specifically, the sentence "there is a time after which $PROGRESS[k]$ does no longer increase" has to be replaced by the sentence "'there is a time after which $PROGRESS[k, i]$ remains forever equal to $LAST[k, i]$".) As they are very close to the previous ones and tedious, we don't detail these proofs. (According to the usual sentence, "They are left as an exercise to the reader".)

```
task T1:
(01)  when leader() is invoked:
(02)    for_each k ∈ candidates_i do
(03)        susp_i[k] ← Σ_{1≤j≤n} SUSPICIONS[j,k] end_for;
(04)    let (−, ℓ) = lex_min({(susp_i[k], k)}_{k∈candidates_i});
(05)    return(ℓ)

task T2:
(06)  repeat_forever
(07)    while (leader() = i) do
(08.R1)       for_each k ∈ {1,...,n} \ {i} do
(08.R2)          if (PROGRESS[i,k] = LAST[i,k]) then
(08.R3)             PROGRESS[i,k] ← ¬LAST[i,k] end_if
(08.R4)       end_for;
(09)          if STOP[i] then STOP[i] ← false end_if
(10)    end_while;
(11)    if (¬ STOP[i]) then STOP[i] ← true end_if
(12)  end_repeat

task T3:
(13)  when timer_i expires:
(14)    for_each k ∈ {1,...,n} \ {i} do
(15)      stop_k_i      ← STOP[k];
(16.R1)   progress_k_i ← PROGRESS[k,i];
(17.R1)   if (progress_k_i ≠ LAST[k,i]) then
(18)        candidates_i ← candidates_i ∪ {k};
(19.R1)     LAST[k,i] ← progress_k_i
(20)      else_if (stop_k_i) then
(21)        candidates_i ← candidates_i \ {k}
(22)      else_if (k ∈ candidates_i) then
(23)        SUSPICIONS[i,k] ← SUSPICIONS[i,k] + 1;
(24)        candidates_i ← candidates_i \ {k}
(25)      end_if
(26)    end_for;
(27)    set timer_i to max({SUSPICIONS[i,k]}_{1≤k≤n})
```

Figure 3. All variables are 1WMR and bounded (code for p_i)

The same reasoning as the one done in the proof of the Theorem 2 shows that each shared variable $SUSPICIONS[j,k]$, $\leq j,k \leq n$, is bounded. Combined with the fact that the variables $PROGRESS[j,k]$ and $LAST[j,k]$ are boolean, we obtain the following theorem.

Theorem 6 *All the variables used in the algorithm described in Figure 3 are bounded.*

The following theorem is the counterpart of Theorem 3.

Theorem 7 *Let p_ℓ be the process elected as the eventual common leader, and p_i, $i \neq \ell$, any correct process. There is a time after which the only variables that may be written are $PROGRESS[\ell,i]$ (written by p_ℓ) and $LAST[\ell,i]$ (written by p_i).*

Proof The proof that the variables $PROGRESS[\ell,j]$, $1 \leq j \leq n$, are infinitely often written, and the proof that there is a time after which the variables $STOP[j]$, $1 \leq j \leq n$, and the variables $SUSPICIONS[j,k]$, $1 \leq j,k \leq n$, are no longer written is the same as the proof done in Theorem 3.

The fact that there is a time after which $PROGRESS[x,j]$, $1 \leq x,j \leq n$, $x \neq \ell$, are no longer written follows from the fact that, after p_ℓ has been elected, no process p_x executes the body of the **while** loop of task $T2$.

Let us now consider any variable $LAST[x,y]$, $x \neq \ell$. As, after p_ℓ has been elected, no correct process p_x, $x \neq \ell$, updates $PROGRESS[x,y]$ (at line 08.R2), it follows that there is a time after which $LAST[x,y] = PROGRESS[x,y]$ remains forever true for $1 \leq x,y \leq n$ and $x \neq \ell$. Consequently, after a finite time, the test of line 17.R1 is always false for p_x, $x \neq \ell$, and $LAST[x,y]$ is no longer written. $\square_{Theorem\ 7}$

Finally, the next theorem follows directly from Corollary 1.

Theorem 8 *The Ω algorithm described in Figure 3 is optimal with respect to the number of processes that have to write the shared memory.*

5 Conclusion

This paper has addressed the problem of electing an eventual leader in an asynchronous shared memory system. It has three main contributions.

- The first contribution is the statement of an assumption (a property denoted AWB) that allows electing a leader in the shared memory asynchronous systems that satisfy that assumption. This assumption requires that after some time (1) there is a process whose write accesses to some shared variables are timely, and (2) the other processes have asymptotically well-behaved timers. The notion of asymptotically well-behaved timer is weaker than the usual notion of timer where the timer durations have to monotonically increase when the values to which they are set increase. This means that AWB is a particular weak assumption.
- The second contribution is the design of two algorithms that elect an eventual leader in any asynchronous shared memory system that satisfies the assumption AWB. In addition of being independent of t (the maximum number of processes allowed to crash), and being based only on one-writer/multi-readers atomic shared variables, these algorithms enjoy noteworthy properties. The first algorithm guarantees that (1) there is a (finite) time after which a single process writes forever the shared memory, and (2) all but one shared variables have a bounded domain. The second algorithm uses (1) a bounded memory but (2) requires that each process forever writes the shared memory.

- The third contribution shows that the previous trade-off (bounded/unbounded memory vs number of processes that have to write) is inherent to the leader election problem in asynchronous shared memory systems equipped with AWB. It follows that both algorithms are optimal, the first with respect to the number of processes that have to forever write the shared memory, the second with respect to the boundedness of the memory.

Several questions remain open. One concerns the first algorithm. Is it possible to design a leader algorithm in which there is a time after which the eventual leader is not required to read the shared memory? Another question is the following: is the second algorithm optimal with respect to the size of the control information (bit arrays) it uses to have a bounded memory implementation?

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 PODC* pp. 328-337, 2004.

[2] Boichat R., Dutta P., Frølund S. and Guerraoui R., Deconstructing Paxos. *ACM Sigact News, Distributed Computing Column*, 34(1):47-67, 2003.

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

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

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

[6] Fernández A., Jiménez E. and Raynal M., Electing an Eventual Leader in an Asynchronous Shared Memory System. *Tech Report #1821*, 18 pages, Université de Rennes, France, November 2006.

[7] 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.

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

[9] 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.

[10] Guerraoui R., Kapalka M. and Kouznetsov P., The Weakest failure Detectors to Boost Obstruction-Freedom. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer-Verlag LNCS #4167, pp. 376-390, 2006.

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

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

[13] 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.

[14] Hélary J.-M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE TPDS*, 11(9):897-909, 2000.

[15] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[16] Herlihy M.P., Luchangco V., Moir M. and Scherer III W.N., Software Transactional Memory for Dynamic Sized Data Structure. *Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'03)*, pp. 92-101, 2003.

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

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

[19] Larrea M., Fernández A. and Arévalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Resilient Distributed Systems (SRDS'00)*, pp. 52-60, 2000.

[20] Lo W.-K. and Hadzilacos V., Using failure Detectors to solve Consensus in Asynchronous Shared Memory Systems. *Proc. 8th Int'l Workshop on Distributed Computing (WDAG'94)*, Springer Verlag LNCS #857, pp. 280-295, 1994.

[21] Malkhi D., Oprea F. and Zhou L., Ω Meets Paxos: Leader Election and Stability without Eventual Timley Links. *Proc. 19th Int'l Symposium on DIStributed Computing (DISC'05)*, Springer Verlag LNCS #3724, pp. 199-213, 2005.

[22] 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.

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

[24] Mostéfaoui A., Raynal M. and Travers C., Crash Resilient Time-Free Eventual Leadership. *Proc. 23th IEEE Symposium on Reliable Dists. Systems*, pp. 208-218, 2004.

[25] 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.

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

[27] Raynal M., A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *ACM SIGACT News, Distributed Computing Column*, 36(1):53-70, 2005.