Algorithmica (2010) 56: 550–576
DOI 10.1007/s00453-008-9190-2

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

Antonio Fernández · Ernesto Jiménez · Michel Raynal · Gilles Trédan

Received: 22 May 2007 / Accepted: 26 March 2008 / Published online: 19 April 2008

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*) is particularly weak. It is made up of two complementary parts. More precisely, it requires that, after some time, (1) there is a process whose write accesses to some shared variables be timely, and (2) the timers of $(t - f)$ other processes be asymptotically well-behaved (t denotes the maximal number of processes that may crash, and f the actual number of process crashes in a run). 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

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 and Gilles Trédan was supported by the European Network of Excellence ReSIST.

A. Fernández
LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain
e-mail: anto@gsyc.es

E. Jiménez
EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain
e-mail: ernes@eui.upm.es

M. Raynal (✉) · G. Trédan
IRISA, Université de Rennes, Campus de Beaulieu, 35042 Rennes, France
e-mail: raynal@irisa.fr

G. Trédan
e-mail: gtredan@irisa.fr

(a timer works incorrectly when it expires at arbitrary times, i.e., independently of the value it has been set to).

The paper then focuses on the design of t-resilient *AWB*-based eventual leader protocols. "t-resilient" means that each protocol can cope with up to t process crashes (taking $t = n - 1$ provides wait-free protocols, i.e., protocols that can cope with any number of process failures). Two protocols are presented. The first 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 protocol is consequently optimal with respect to the number of processes that have to write the shared memory. The second protocol guarantees that all the shared variables have a bounded domain. This is obtained at the following additional price: $t + 1$ processes are required to forever write the shared memory. A theorem is proved which states that this price has to be paid by any protocol that elects an eventual leader in a bounded shared memory model. This second protocol 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 protocols show an inherent tradeoff relating the number of processes that have to write the shared memory and the bounded/unbounded attribute of that memory.

Keywords Asynchronous system · Atomic register · Eventual leader · Fault-tolerance · Omega · Process crash · Shared memory · System model · Timer property · Timing assumptions · t-resilient protocol

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. One of the most famous protocol based on such an eventual leader service is the well-known state machine replication protocol called Paxos [19]. An eventual leader service (also called *unreliable failure detector* or *distributed oracle* [5, 30]) is usually denoted Ω in the literature [6].

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 [3, 27]). Of course, considering a synchronous system would allow designing an eventual leader service, but this is not sensible as this is a very strong assumption on the system behavior. So, a central issue consists in finding timing assumptions that are, at the same time, "strong enough" in order a leader service can be built, and "weak enough" in order that they are "practically" meaningful (i.e., they are satisfied nearly always [29]). Finding such necessary and sufficient assumptions

remains a fundamental issue from both a practical and theoretical points of view. Seen from a theory point of view, the answer would establish the asynchrony boundary beyond which the problem cannot be solved. Seen from a practical point of view, the answer would define the requirements a system has to satisfy in order to solve the problem, and would consequently provide the engineers with the minimal requirements their underlying systems have to meet.

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 [1, 4, 10, 21]. The Ω protocols presented in this paper are suited to such systems. Examples of shared memory Ω-based protocols can be found in [9, 14].

On another side, 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 [11] that Ω constitutes the weakest *contention manager* that allows transforming any obstruction-free [16] software transactional memory into a non-blocking transactional memory [17]. 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.

Content of the Paper This paper is on the design of protocols that construct an eventual leader service Ω in an asynchronous shared memory system where processes can crash. Let n be the total number of processes, and t the maximal number of processes that can crash in a run. We are interested in the design of t-resilient protocols, i.e., protocols that can cope with up to t process crashes. This means that the protocol has to work correctly when no more than t processes are faulty. When, more than t processes are faulty, the protocol is allowed to behave arbitrarily. When, $t = n - 1$, a t-resilient protocol is also called a *wait-free* protocol [15]. Usually, the system parameter t is explicitly used in the text of a t-resilient protocol. As, in practice, the number of processes that crash in a given run is very small, it is interesting to design t-resilient protocols. Let f, $0 \leq f \leq t$, denote the number of processes that crash in a given run. The paper has three main contributions.

Contribution #1 The paper first proposes a behavioral assumption for the asynchronous system be able to implement an eventual leader, that is particularly weak. It is made up of two matching parts. In each run, there are a finite (but unknown) time τ, and a process p that does not crash in that run (p is not a priori known) such that, after τ:

- If $f < t$, 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
- There are $(t - f)$ correct processes q, $q \neq p$, that have 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 of $n - (t - f)$ correct processes can behave arbitrarily, i.e., they can expire at times that are arbitrary with respect to the values they have been set to. Moreover, the timers of the $(t - f)$ correct processes involved in the additional assumption can behave arbitrarily during arbitrarily long (but finite) periods. Moreover, as we will see in their formal definition, their durations are not required to monotonically increase when their timeout values increase. They only have, after some time, to be lower-bounded by some monotonically increasing function.

It is worth noticing that no process (but p) is required to have a synchronous behavior, and only some timers have to eventually satisfy a weak behavioral property. Moreover, it is easy to see that, in the runs where $f = t$, the previous assumption is always trivially satisfied despite asynchrony (no process is required to behave synchronously, and no timer is required to behave correctly).

Contribution #2 The paper then presents two t-resilient protocols that construct an eventual leader service Ω in all the runs that satisfy the previous behavioral assumptions. Both protocols use one-writer/multi-readers (1WMR) atomic registers.

- In the first protocol, all the shared registers (but one) have a bounded domain. More specifically, this means that, be the run finite or infinite, there is a time after which only one shared register keeps on increasing. Interestingly, all the timeout values stop increasing.

  Moreover, there is a single process that writes forever the shared memory. The protocol is consequently *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 t-resilient protocol improves the first one in the sense that all the shared registers used by the processes to communicate are bounded. This nice property is obtained by using two boolean flags and a simple hand-shaking mechanism between each pair of processes. For each ordered pair of processes (p, q), these flags allow, in one direction, p to pass an information to q, and in other direction, q to inform p that it has read that information.

Interestingly, the design of both protocols is based on simple ideas. Moreover, these protocols are presented in an incremental way: the second t-resilient protocol is designed as a simple improvement of the first one. This makes easier both its understanding and its proof.

Contribution #3 The paper proves lower bound results for the considered computing model. These results concern the minimal number of processes that have to write the shared memory when that memory is not bounded and when it is bounded, and the minimal number of processes that have to read the shared memory.

More precisely, three theorems are stated and proved. The first shows that the process that is eventually elected has to forever write the shared memory. Another theorem shows that any process (but the eventual leader) has to forever read the shared memory. Finally, the last theorem shows that, if the shared memory is bounded, then $t + 1$ processes have to forever write the shared memory. These theorems show that the two t-resilient protocols presented in the paper are optimal with respect to these criteria.

Related Work in the Message-Passing Context The design of protocols that implement an eventual leader service has received a lot of attention in the message-passing context, i.e., when the processes cooperate by exchanging messages through an underlying network. The implementation of Ω in asynchronous message-passing systems is an active research area. Two main approaches have been investigated: the *timer*-based approach and the *message pattern*-based approach.

The timer-based approach relies on the addition of timing assumptions [7]. 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 protocols implementing Ω in such "augmented" asynchronous systems are based on timeouts (e.g., [2, 3, 20]). 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 [2] 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 $\diamondsuit t$-*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 $\diamondsuit t$-*source* is not known in advance and may never be explicitly known. It is also shown in [2] that there is no leader protocol if the system has only $\diamondsuit(t-1)$-*sources*. A versatile adaptive timer-based approach has been developed in [23].

The message pattern-based approach, introduced in [25], 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 $\diamondsuit t$-*source* assumption. This approach has been applied to the construction of an Ω protocol in [27].

A *hybrid* protocol that combines both types of assumption is developed in [28]. More precisely, this protocol 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 [29].

Related Work in the Shared Memory Context To our knowledge, only three eventual leader protocols suited to the shared memory context have been proposed so far [8, 13]. The protocol presented in [13] assumes that there is a finite time after which all the processes behave synchronously. So, this timing assumption is pretty strong.

The second paper [8] investigates an assumption that is at the origin of the assumption presented in this paper. The algorithms in [8] actually present our early

work on the election of an eventual leader in an asynchronous shared memory system. That preliminary work considered only the case $t = n - 1$ (namely, the wait-free case), and presents two wait-free algorithms (without their proofs). As in this paper, one of these algorithms considers that all but one variables are bounded, while the other algorithm addresses the case where all the variables are bounded. Both these algorithms are different from their counterparts presented here: they are less general (they are not t-resilient), use more shared variables and are less efficient.[1] Moreover, the assumption used here is slightly weaker and more general than the one introduced in [8]. It is also important to notice that the t-resilient algorithms presented here are original, and do not result from a "simple generalization" of the algorithms presented in [8]. Not only they use less shared variables (as already mentioned), but their design is based on different principles. Finally, when considering $t = n - 1$, these t-resilient algorithms are more efficient than their wait-free counterparts presented in [8].

Roadmap The paper is made up of 5 sections. Section 2 presents the system model and the additional behavioral assumption. Then, Sects. 3 and 4 present in an incremental way the two t-resilient protocols implementing an eventual leader service, and show they are optimal with respect to the number of processes that have to write or read the shared memory. Finally, Sect. 5 provides concluding remarks.

2 System Model, Eventual Leader, and Additional Assumption

2.1 Base Asynchronous Shared Memory Model

The system consists of n ($n > 1$) processes denoted $p_1, \ldots, p_n$. We assume that process identities are all different and totally ordered. Hence, for simplicity we make the integer i to denote 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 protocol. 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 ($1 \leq t \leq n - 1$),[2] 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 (1WMR). "1WMR" means that a single process can write into it, but all the

[1] As an example, the task $T3$ in these algorithms (see Figs. 2 and 4) considers only the value k, where p_k is the current leader, while all the values $k \neq i$ have to be considered in the algorithms presented in [8].

[2] 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 $t = n - 1$. Let nevertheless observe that, in practice, failures are rare, so small values of t (with respect to n) are realistic.

processes can read it. (Let us observe that using 1WMR atomic registers is particularly suited for cache-based distributed shared memory.[3]) The only process allowed to write an atomic register is called its owner. *Atomic* means that, although read and write operations can take time and overlap, everything appears as (1) if the read and write operations are executed one after the other, (2) each operation appearing as if it has been executed instantaneously at some point of the time line between its invocation and return events (this is called the *linearization* point of the operation [18]). Atomicity is a very powerful conceptual tool that allows us to think and reason as if the operations are totally ordered and take no time to execute.

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.

In the following we consider that some shared registers are *critical*, while the other shared registers are not critical. A critical register is a register whose accesses by some processes have to satisfy some timing constraints (see Sect. 3.2).[4]

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* shared memory model where up to t processes may crash. 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 [6]. (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.

[3] As observed in the Introduction 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 [10, 21]. Protocols based of commodity disks are described in [9, 14].

[4] The notion of critical register is not necessary to implement an eventual leader. All shared registers could implicitly be considered as being critical. The *critical* attribute is used only to restrict the set of registers involved in the additional assumptions required to elect an eventual leader.

The Ω leader abstraction has been formally introduced in [6]. It has been shown to be weakest, in terms of information about failures, to solve consensus in asynchronous systems prone to process crashes, be these systems message-passing systems [6] or shared memory systems [22]. Several consensus protocols based on such an eventual leader service have been proposed (e.g., [12, 19, 26] for message-passing systems, and [9, 14] 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 *AWB* (for *a*symptotically *w*ell-*b*ehaved). The resulting system is consequently denoted $\mathcal{AS}_{n,t}[AWB]$.

Each process p_i is equipped with a timer denoted $timer_i$. The intuition that underlies *AWB* is that, once a process p_ℓ that has not crashed 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 be "eventually synchronous" [5, 7] would work but is much more constraining than what is necessary. The aim of the *AWB* additional assumption is to state constraints that allow implementing Ω while being "as weak as possible". "As weak as possible" is an intuitive notion, different from the "weakest possible" formal notion. It means that, when one wants to implement Ω in a shared memory system, as far as we are concerned, we know neither an assumption weaker than *AWB*, nor the answer to the question: "Is *AWB* the weakest additional assumption?". 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 and AWB_2 are "matching" properties. AWB_1 is on the existence of a process whose behavior has to satisfy a synchrony property. AWB_2 is on the timers of a subset of the other processes; it states a property that allows these processes to perceive the progress of the process involved in AWB_1.

The Assumption AWB_1 That assumption restricts the asynchronous behavior of one process. Given a run characterized by a value of f, it is defined as follows.

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

Let us first observe that this assumption is always satisfied when $f = t$. When $f < t$, it 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 AWB_1" to say that p_ℓ is a process that makes true that assumption.

The Assumption AWB_2 The definition of AWB_2 involves timers and relies on the notion of *asymptotically well-behaved* timer. The aim of that notion is to capture timer behaviors that are sufficient to implement an eventual leader but could be too weak to solve other problems. From an operational point of view, the intuition that underlies that notion is that there is a time τ after which, whatever the duration δ and the time $\tau' \geq \tau$ at which it is set to δ, that timer expires after some finite time τ'' such that $\tau'' \geq \tau' + \delta$. That is the only constraint on the timer expiration for that timer to be asymptotically well-behaved. If the timer is set to $\delta 1$ at some time $\tau 1 \geq \tau$ and expires at $\tau 1'$, and the same or another timer is set to $\delta 2 > \delta 1$ at some time $\tau 2 \geq \tau$ and expires at $\tau 2'$, it is not required that $\tau 2' - \tau 2 > \tau 1' - \tau 1$.

In order to formally define the notion of asymptotically well-behaved timer, we first introduce a function $f : \mathbf{R}^+ \times \mathbf{R}^+ \to \mathbf{R}^+$, 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. Its monotonicity properties are the following. There are two (possibly unknown) bounded values x_{AWB_2} and τ_{AWB_2} such that:

(f1) $\forall \tau_2, \tau_1 : \tau_2 \geq \tau_1 \geq \tau_{AWB_2}, \forall x_2, x_1 : x_2 \geq x_1 \geq x_{AWB_2}$: $f(\tau_2, x_2) \geq f(\tau_1, x_1)$. (After some point, $f()$ is not decreasing with respect to τ and x).

(f2) $\lim_{x \to +\infty} f(\tau_{AWB_2}, x) = +\infty$. (Eventually, $f()$ always increases.[5])

Thanks to the function $f()$, we are now in order to give a general and precise definition for 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 the run R, if there is a function $f_R()$, as defined above, such that:

(f3) $\forall \tau : \tau \geq \tau_{AWB_2}, \forall x : x \geq x_{AWB_2}$: $f_R(\tau, x) \leq T_R(\tau, x)$.

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 $(\tau_{AWB_2}, x_{AWB_2})$, 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 Fig. 1.) As we can see, the notion of "asymptotically well-behaved" limits the inaccuracy of a timer, it does not require it to be "perfect" (i.e., to expire exactly when the duration it has been set to has elapsed).

Practically, the property (f3) means that an asymptotically well-behaved timer is allowed to expire at arbitrary times (i.e., times that are unrelated to the timeout values it has been set to) during an arbitrary but finite time, after which it behaves correctly in

[5]If the image of $f()$ is the set of natural numbers, then this condition can be replaced by $x_2 > x_1 \Rightarrow f(\tau_{AWB_2}, x_2) > f(\tau_{AWB_2}, x_1)$.

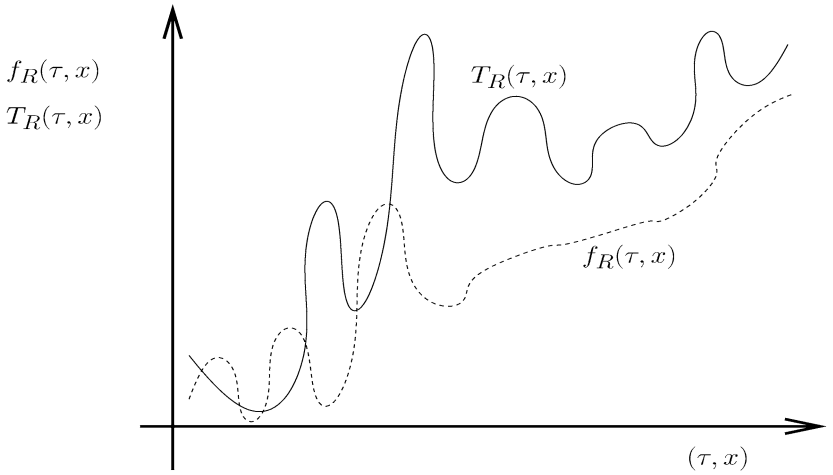

Fig. 1 $T_R()$ asymptotically dominates $f_R()$

the sense that it never expires "too early" (without being required to behave monotonically according to the durations is it set to). Moreover, there is no upper bound on the duration after which it expires, except that this duration is finite.

To motivate and illustrate the previous discussion, let us consider the case of a system with computers whose local clocks suffer drifts, and are periodically resynchronized (e.g., by means of the Network Time Protocol (NTP) [24]). In such a system, it can happen that the timer of a processor p_i is set to a duration x before the resynchronization of p_i's local clock, and timeouts after that resynchronization. In that case, the timer can expire at a time that is not correctly related to x. More generally, for the same duration x, setting a timer at a time τ can lead to a larger sleeping time (for the timer) than setting the timer at a time τ', $\tau' > \tau$, with the same x. This in fact can occur for infinite triples x, τ, and τ'. Hence, the associated function $T_R()$ is not monotonically increasing. But, $T_R(\tau, x)$ can be lower-bounded by a function $f_R(\tau, x)$, which satisfies the properties (f1), (f2), and (f3) defined above (the definition of $f_R(\tau, x)$ has then to take into account the maximal drifts of the local clocks).

A timer that does not asymptotically well-behave can be seen as a kind of Byzantine timer (i.e., a timer that, whatever the duration it has been set to, expires at arbitrary times). Such a bad behavior can be due, for example, to erratic clock drifts and resynchronizations (as analyzed before) or to cosmic rays that modify bits of a timeout value. The notion of "asymptotically well-behaved" shows that a timer has not to be "perfect" in order to be useful when implementing an eventual leader. It is not required to behave monotonically (with respect to the durations it is set to) as long as its behavior is lower-bounded by a function $f()$ satisfying the properties previously stated. We are now in order to state the assumption AWB_2. It is the following.

AWB_2: The timers of $(t - f)$ correct processes (different from the process p_ℓ that satisfies AWB_1) are asymptotically well-behaved.

When we consider *AWB*, it is important to notice that any process (but p_ℓ, which is 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. Moreover, the timers of up to $(n-t)+f$ correct processes can behave arbitrarily. This means that, in the runs where $f=t$, the timers can behave arbitrarily. It follows that the timing assumption *AWB* is particularly weak.

In the following we say "p_x is involved in AWB_2" to say that p_x is a correct process that has an asymptotically well-behaved timer.

3 A Write-Optimal t-Resilient Protocol for $\mathcal{AS}_{n,t}[AWB]$

3.1 Principle of the Protocol

The first t-resilient protocol that implements an eventual leader in $\mathcal{AS}_{n,t}[AWB]$ is described in Fig. 2. It 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 its current leader 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 AWB_1 if p_k satisfies that assumption). So, when it has to determine its current leader, p_i does not consider the whole set of suspicions (the array $SUSPICIONS[1..n, 1..n]$), but only an appropriate part of it. More precisely, for each process p_k, p_i takes into account only the $(t+1)$ entries with the smallest values among the n counters $SUSPICIONS[1,k], \ldots, SUSPICIONS[n,k]$. As we will see, due AWB_2, this allows it to eliminate the erroneous suspicions and consequently determine a correct eventual common leader.

As several processes can be equally suspected, p_i uses the function $lex\_min(X)$ 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 a size n vector plus a $n \times n$ matrix of 1WMR shared atomic registers.

- $PROGRESS[1..n]$ is an array of 1WMR shared integer variables. Only p_i can write $PROGRESS[i]$. In order to indicate to the other processes that it is still alive, p_i regularly increases $PROGRESS[i]$ when it considers it is the leader.
- $SUSPICIONS[1..n, 1..n]$ is an array of shared variables that contain non-negative integers. The entries of the vector $SUSPICIONS[i, 1..n]$ can be written only by p_i. Intuitively, $SUSPICIONS[i,j] = x$ means that, up to now, the process p_i has suspected $x-1$ times the process p_j to have crashed.

task $T1$:
(1) **when** leader() **is invoked**:
(2) **for_each** $k \in \{1, ..., n\}$ **do**
(3) **let** $witness_i[k]$ = set of $(t+1)$ process identities such that
$\forall x \in witness_i[k],\ \forall y \notin witness_i[k]$: $(SUSPICIONS[x,k],x) < (SUSPICIONS[y,k],y)$;
(4) **let** $susp_i[k] = \Sigma_{x \in witness_i[k]}\ SUSPICIONS[x,k]$
(5) **end_for**;
(6) return(ℓ) **where** ℓ **is such that** $(-,\ell) = lex\_min(\{(susp_i[k],k)\}_{1 \le k \le n})$

task $T2$:
(7) **repeat_forever**
(8) **let** $my\_witnesses_i$ = set of $(t+1)$ process identities such that
$\forall x \in my\_witnesses_i, \forall y \notin my\_witnesses_i$: $(SUSPICIONS[x,i],x)$
$< (SUSPICIONS[y,i],y)$;
(9) **let** $susp\_count_i = \Sigma_{x \in my\_witnesses_i}\ SUSPICIONS[x,i]$;
(10) **if** $\big((\text{leader}() = i) \vee (susp\_count_i \neq prev\_susp\_count_i)\big)$
(11) **then** $progress_i \leftarrow progress_i + 1$; $PROGRESS[i] \leftarrow progress_i$
(12) **end_if**;
(13) $prev\_susp\_count_i \leftarrow susp\_count_i$
(14) **end_repeat**

task $T3$:
(15) **when** $timer_i$ **expires**:
(16) $k \leftarrow$ leader();
(17) **let** $witness\_k_i$ = set of $(t+1)$ process identities such that
$\forall x \in witness\_k_i,\ \forall y \notin witness\_k_i$: $(SUSPICIONS[x,k],x) < (SUSPICIONS[y,k],y)$;
(18) **let** $susp\_k_i = \Sigma_{x \in witness\_k_i}\ SUSPICIONS[x,k]$;
(19) **if** $\big((k \neq i) \wedge (i \in witness\_k_i) \wedge (k = prev\_ld_i) \wedge (susp\_k_i = prev\_susp_i)\big)$
(20) **then** $progress\_k_i \leftarrow PROGRESS[k]$;
(21) **if** $(progress\_k_i \neq last_i[k])$
(22) **then** $last_i[k] \leftarrow progress\_k_i$
(23) **else** $suspicions_i[k] \leftarrow suspicions_i[k] + 1$;
(24) $SUSPICIONS[i,k] \leftarrow suspicions_i[k]$
(25) **end_if**
(26) **end_if**;
(27) $prev\_ld_i \leftarrow k$; $prev\_susp_i \leftarrow susp\_k_i$;
(28) $timeout_i \leftarrow susp\_k_i$; set $timer_i$ to $timeout_i$

Fig. 2 t-resilient eventual leader election with all variables bounded, but $PROGRESS[\ell]$ (code for p_i)

The variables $PROGRESS[k]$, $1 \le k \le n$, are the only *critical* variables of the algorithm. This means that they are the only shared variables concerned by the additional assumption AWB_1. Said differently, there is no timing constraint on the write accesses to the shared variables $SUSPICIONS[k,\ell]$, $1 \le k, \ell \le n$.

To achieve correctness, the initial values of the previous shared variables could be arbitrary.[6] However, to make the presentation easier, improve efficiency,

[6]This means that the protocol 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), the statement "set

and reach optimality in some cases, we consider in the following that initially $SUSPICIONS[i, j] = 1, 1 \leq i, j \leq n, i \neq j$, and $SUSPICIONS[i, i] = 0, 1 \leq i \leq n$.

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; $suspicions_i[k]$ is used to update $SUSPICIONS[i, k]$.
- $timeout_i$ contains the latest timeout value used by p_i to set its timer $timer_i$.
- $susp\_count_i$ is a variable used to count the current number of meaningful suspicions of p_i (issued by the other processes); $prev\_susp\_count_i$ is used to keep the previous value of $susp\_count_i$.
- $progress\_k_i$, $witness\_k_i$, $susp\_k_i$, $prev\_ld_i$, $prev\_susp_i$, $my\_witnesses_i$, $witness_i[1..n]$, and $susp_i[1..n]$ are auxiliary local variables used by p_i.

Additionally, as said above, process p_i has a timer $timer_i$ which is initially set with some arbitrary timeout value.

3.3 Process Behavior

The behavior of a process p_i is described in Fig. 2. It is decomposed in three tasks.

*Task T*1 The first task (lines 1–6) defines the way the current leader is determined. For each process p_k, p_i first computes the number of relevant suspicions that concern p_k. As already mentioned, those are defined by the $(t + 1)$ entries of the vector $SUSPICIONS[1..n, k]$ with the smallest values (lines 3–4). The $(t + 1)$ processes whose entries in $SUSPICIONS[1..n, k]$ have the smallest values are called the *witness* processes for p_k. The current leader is then defined as the process that is currently the least suspected, when considering only the relevant suspicions, i.e., for each process p_k, the suspicions issued by its witness processes (line 6).

*Task T*2 The second task (lines 7–14) is an infinite loop that is on the management of the shared variable $PROGRESS[i]$ (line 11). More explicitly, a process p_i increases $PROGRESS[i]$ when it considers that it is the leader (test $\mathsf{leader}() = i$, line 11), or when the number of its relevant suspicions has changed since the last time it has executed that task (test $susp\_count_i \neq prev\_susp\_count_i$, line 11; this means that p_i has been considered as a leader since its last execution of $T2$). This allows p_i to inform the processes that suspected it that it is still alive.

$timer_i$ to $timeout_i$" (line (28) of Fig. 2) has to be replaced by "set $timer_i$ to $\max(timeout_i, 1)$" in order a timer be always set to a positive value.

*Task T*3 The third task is associated with p_i's timer expiration. It is where p_i possibly suspects the current leader and where it sets its timer ($timer_i$).

1. Suspicion management part (lines 16–27). First, p_i determines its current leader p_k (line 16) and the current set of $(t+1)$ processes that suspect the least p_k; these processes define the set $witness\_k_i$ (line 17). A process p_i is allowed to worry about its current leader p_k (line 19) only if (1) it does not consider itself as the current leader (i.e., $i \neq k$), (2) it belongs to the set of p_k's witnesses ($i \in witness\_k_i$), and, (3) at the previous timer expiration, p_k was its leader ($k = prev\_ld_i$) and, since that time, p_k has not seen an increase in its number of relevant suspicions ($susp\_k_i = prev\_susp_i$). The predicate $(k = prev\_ld_i) \wedge (susp\_k_i = prev\_susp_i)$ allows p_i to check if p_k was continuously the leader between two consecutive expirations of $timer_i$. So, when the predicate of line 19 is satisfied, p_i reads the value of $PROGRESS[k]$ (line 20) to see if that variable has been increased since the latest time it read it (line 21). If it is the case, p_i updates $last_i[k]$ accordingly (line 22). If it is not the case, p_i suspects once more p_k (line 24). As we can see, in order to check if its leader p_k is alive or in order to suspect it, a process p_i has to currently be one of the witness of p_k. Finally, p_i updates the values of $prev\_ld_i$ and $prev\_susp_i$ (line 27).
2. Timer setting part (line 28). Then, p_i resets its timer to an appropriate timeout value. That value is the number of current relevant suspicions that has been computed in $susp\_k_i$. Let us observe that, if the leader does not change and the number of its relevant suspicions does no longer increase, $timeout_i$ keeps forever the same value.

3.4 Proof of Correctness

Let us consider a run R of the protocol described in Fig. 2 in which the assumptions AWB_1 and AWB_2 defined in Sect. 2.3 are satisfied. This section shows that an eventual leader is elected in that run. The proof is decomposed into several lemmas. The first lemma shows that faulty processes eventually stop suspecting all processes.

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 modify shared variables. □

The following lemma shows that all processes with well-behaved timers eventually stop suspecting the "partially synchronous" process.

Lemma 2 *Assuming $f < t$, let p_i be a correct process involved in AWB_2 (i.e., its timer is eventually well-behaved), and p_j a correct process that satisfies AWB_1. Then, $SUSPICIONS[i, j]$ is bounded.*

Proof The proof breaks down into two cases, depending on whether variable $PROGRESS[j]$ is perpetually increased by p_j. Intuitively, this lemma says that if

p_j stops increasing that variable, it is no longer leader, and consequently the other processes will not suspect it. If, otherwise, p_j increases the variable permanently, we show that eventually p_j does so at least once between two read operation from p_i, which prevents it from suspecting p_j.

Let S be the sequence of updates of $PROGRESS[j]$ issued by p_j. Let us observe that all these updates are issued by the task $T2$ of p_j (line 11). We consider two cases.

- S is finite.[7]

  In that case, there is a finite time τ after which the predicate $(\mathsf{leader}() = j) \vee (susp\_count_j \neq prev\_susp\_count_j)$ evaluated by p_j (line 10) is always false. It follows from this observation and line 13 that, after τ, the local predicate $susp\_count_j = prev\_susp\_count_j$ remains permanently true.

  Assume now, by way of contradiction, that $SUSPICIONS[i, j]$ never stops increasing. Then, from the above local predicate, there is a time τ' after which process p_i is never among the $(t + 1)$ witnesses of p_j. (These witness processes are defined at line 17.) Note that the condition at line 19 forces that in order to increase $SUSPICIONS[x, j]$ (line 24), a process p_x has to consider itself as one of the $(t + 1)$ witnesses of p_j. We conclude that, after τ', $SUSPICIONS[i, j]$ is never increased.
- S is infinite.

  Due to the assumption AWB_1, there are a time τ_{AWB_1}, and a bound Δ such that, after τ_{AWB_1}, any two consecutive updates of $PROGRESS[j]$ by p_j are completed by at most Δ time units.

  By assumption AWB_2, the timer of p_i is asymptotically well-behaved, which means that, for each run R, there are a function $f_R()$ and parameters τ_{AWB_2} and x_{AWB_2}. Let $x_0 \geq x_{AWB_2}$ be a finite value such that $f_R(\tau_{AWB_2}, x_0) = \Delta' > \Delta$. Assumption (f2) implies that such a value x_0 does exist.

  All the time instants considered in the following are after $\max(\tau_{AWB_1}, \tau_{AWB_2})$. Let us assume (by contradiction) that $SUSPICIONS[i, j]$ increases forever.

  1. As $SUSPICIONS[i, j]$ increases forever (line 24), it follows that p_i is a witness of p_j infinitely often (test of line 19), which means that $SUSPICIONS[i, j]$ is infinitely often one of the $(t + 1)$ smallest value in the vector $SUSPICIONS[1..n, j]$. We conclude that there is a time τ' after which $susp\_k_i > x_0$ (lines 17 and 18). Consequently, after τ', any two successive expirations of $timer_i$ are separated by at least Δ' time units (line 28).
  2. As, just before $SUSPICIONS[i, j]$ is increased, the predicate $(j = prev\_ld_i) \wedge (susp\_k_i = prev\_susp\_ld_i)$ is true (test of line 19), it follows that, during at least Δ' time units, p_j has been the leader without being demoted. (The fact that p_j has not been demoted follows from the following observation. As the $SUSPICIONS[x, y]$ variables can only increase, we can conclude from $susp\_k_i = prev\_susp\_ld_i$ that the number of relevant suspicions of p_j have not increased and consequently p_j has been continuously the leader between the end of the first computation of $susp\_k_i$ -kept in $prev\_susp\_ld_i$- and the beginning of the following computation of $susp\_k_i$.)

[7] In that case, the fact that p_i satisfies AWB_2 and p_j satisfies AWB_1, is irrelevant.

As $\Delta' > \Delta$, AWB_1 is satisfied by p_j, and we are after τ_{AWB_1}, it follows that p_j has increased its critical variable $PROGRESS[j]$ between the any two successive readings of that variable by p_i. It follows that necessarily we then have $last_i[j] \neq PROGRESS[j]$, and the test of line 21 is consequently satisfied. It follows that, after $\max(\tau_{AWB_1}, \tau_{AWB_2}, \tau')$, the variable $SUSPICIONS[i, j]$ can no longer be increased, contradicting the assumption that it increases forever. This completes the proof of the lemma. □

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 (i.e., these values are the number of suspicions issued by the processes that are the witnesses of p_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 $(t - f)$ correct processes involved in the assumption AWB_2 (their timers are asymptotically well-behaved). Then, for each process $p_k \notin S$, let S_k denote the set $S \cup \{p_k\}$. (Let us notice that $|S_k| = t + 1$.)

The following lemma shows that, at any time, for any process p_k not in S there is a process in S whose number of suspicions of p_k is no smaller than the largest number of suspicions of p_k's witnesses.

Lemma 3 *Let p_k be a process that does not belong to S. 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. □

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.

In the following lemma we show that, if assumptions AWB_1 and AWB_2 are satisfied, there is at least one process that is suspected a bounded number of times (i.e., the set B is not empty).

Lemma 4 $AWB \Rightarrow (B \neq \emptyset)$.

Proof The proof considers two cases, depending on whether t processes crash or not. If t processes crash, eventually they stop suspecting all correct processes. Since any correct process never suspects itself, at least $t+1$ processes stop suspecting it. Hence all correct processes are in B. If less than t processes crash, then all processes in S stop suspecting the process p_k that satisfies AWB_1. Since p_k never suspects itself, at least $t+1$ processes stop suspecting it, and p_k is in B.

Let us now look in detail at the two cases described above.

- Case $f = t$. Let us first observe that, no process p_k updates $SUSPICIONS[k,k]$. Let us consider a time τ after which the t processes have crashed. Let p_j be any of these processes. It follows from Lemma 1 that, after τ, p_j never updates $SUSPICIONS[j,k]$. Consequently, for any correct process p_k, there are $(t+1)$ entries $SUSPICIONS[1..n,k]$ that are no longer modified after τ. It follows that B is not empty.
- Case $f < t$. Let p_k be the process that satisfies AWB_1. We show that M_k is bounded. Due to Lemma 3, 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 $SUSPICIONS[j,k] = 0$ permanently.
  2. p_j is a faulty process of S_k. $SUSPICIONS[j,k]$ is then bounded due to Lemma 1.
  3. p_j is a process of S_k that is one of the $(t-f)$ correct processes involved in the assumption AWB_2. $SUSPICIONS[j,k]$ is then bounded due to Lemma 2. □

Lemma 5 *There is a time after which any invocation of the primitive* leader() *issued by a process, returns the identity of a process of* B.

Proof The lemma follows from the lines 2–6 and the fact that B is not empty (Lemma 4). □

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

Lemma 6 *There is a single process* p_a *and it is a correct process.*

Proof Let us first observe that $B \neq \emptyset$ (Lemma 4). Moreover, as no two processes have the same identity, there is a single process p_a such that $(M_a, a) = lex\_min(\{(M_x, x) \mid p_x \in B\}$. So, the proof of the lemma consists in showing that p_a is a correct process.

Let assume by contradiction that p_a is a faulty process. This means that there is a time τ_{a1} after which p_a does no longer update $PROGRESS[a]$. As $(M_a, a) = lex\_min(\{(M_x, x) \mid p_x \in B\})$, it follows that, after some time τ_{a2}, p_a is permanently considered leader by all the processes p_i, and consequently, each time its

timer expires, p_i is such that $k = a$ (line 16). Let $\tau \geq \max(\tau_{a1}, \tau_{a2})$. After τ, there is at least one correct process p_i that, each time it executes line 19 is such that $(k = a) \wedge (i \in witness\_k_i)$ (that correct process is not necessarily always the same). Moreover, as $PROGRESS[a]$ remains constant, we then always have $PROGRESS[a] = last_i[k]$. Consequently, infinitely often one of the $(t + 1)$ smallest entries of $SUSPICIONS[1..n, a]$ is increased, contradicting the fact that M_a is bounded. □

Theorem 1 *There is a time after which all the invocations* leader() *return the identity of the same correct process.*

Proof It follows from Lemma 5 that, after some finite time, all the leader() invocations return the identity of a process of B. It follows from lines 2–6 that this identity is the identity a defined in Notation 4. Lemma 6 has shown that p_a is a correct process. □

Theorem 2 *The protocol is write-optimal* (*i.e.*, *after some time a single process writes the shared memory*). *Moreover, be the execution finite or infinite, all variables, but one entry of PROGRESS, are bounded.*

Proof The intuition of the proof is the following. From Theorem 1, a leader is eventually elected. From then on, only the leader updates its entry of *PROGRESS*. Furthermore, eventually the witnesses of the leader do not change and they do not suspect the leader anymore. Since only the entries of the witnesses with respect to the leader can be changed, eventually no entry in $SUSPICIONS[1..n, 1..n]$ changes anymore.

Let us now describe the proof in detail. Let us first consider the array $SUSPICIONS[1..n, 1..n]$. Let τ be the time from which an eventual common leader p_ℓ is elected. Due to Theorem 1 such a time τ does exist. After time τ we have the following.

- As, after τ, any invocation of leader() at line 16 by a process p_i returns always ℓ, we conclude that $\forall i,\ \forall j \neq \ell$, $SUSPICIONS[i, j]$ is never updated after τ (line 24).
- Let τ' be the time from which we have $M_\ell(\tau') = M_\ell$, and $\tau'' = \max(\tau, \tau')$. We now show that no process p_i increases $SUSPICIONS[i, \ell]$ more than once after τ'', which implies that eventually $SUSPICIONS[i, \ell]$ is not updated anymore.

  Let us consider process p_i that evaluates the predicate of line 19 after τ''. We then have $k = \ell$.
  - The predicate is true. In that case, the sub-predicate $i \in witness\_k_i$ is also true. This means that if p_i increased $SUSPICIONS[i, \ell]$, either M_ℓ would be increased or not. The first case contradicts the definition of M_ℓ (namely, $M_\ell = \max(\{M_\ell(\tau)_{\tau \geq 0}\})$). On the other hand, if M_ℓ is not increased, then this implies that p_i stops being a witness of p_ℓ. Therefore, in all further evaluations of line 19 by p_i the predicate will be false.
  - The predicate is false. In that case, it follows directly from the text of the protocol that the shared variable $SUSPICIONS[i, \ell]$ is not updated.

Let us now consider any shared variable $PROGRESS[i]$, $1 \leq i \neq \ell \leq n$. This variable is updated at line 11. After p_ℓ has been elected, the predicate leader() $= i$ is

always false. Moreover, as we have seen previously, there is a time τ' after which no variable $SUSPICIONS[x, y]$ is updated. It follows that, after τ', the predicate $susp\_count_i \neq prev\_susp\_count_i$ is always false. It follows that, there is a time after which no $PROGRESS[i]$ variable, $1 \leq i \neq \ell \leq n$, can be updated; which concludes the proof of the theorem. □

The following corollary is an immediate consequence of Theorem 2 and line 28 of Fig. 2.

Corollary 1 *Be the execution finite or infinite, all the timeout values remain bounded.*

On the Process that is Elected The proof of the protocol relies on the assumption AWB_1 to guarantee that at least one correct process can be elected (i.e., the set B is not empty, Lemma 4, and its smallest pair (M_a, a) is such that p_a is a correct process, Lemma 6). This does not mean that the elected process is a process that satisfies the assumption AWB_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 AWB_2 (its timer is never well-behaved) and p_j does not satisfy AWB_1 (it never behaves synchronously). (A re-reading of the statement of Lemma 2 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 AWB_1, and can consequently suspects these processes infinitely often, and (2) p_j is not synchronous with respect to a process that satisfy AWB_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 AWB_1 in order to guarantee that the set B be never empty.

3.5 Optimality Results

Let $\mathcal{A}$ be a protocol that implements Ω in $\mathcal{AS}_{n,t}[AWB]$. We have the following lower bounds. These bounds are "matching" lower bounds. The intuition that underlies them is simple. The first lower bound (Lemma 7) states that the leader has to write forever the shared memory (this is required for the other processes not to suspect it). The second lower bound (Lemma 8) states that a process has to read the shared memory to perceive the progress of the leader (in order not to suspect it).

Lemma 7 *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.*

Proof Assume, by way of contradiction, that p_ℓ stops writing in the shared memory in run R at time τ. Consider another run R' of $\mathcal{A}$ in which all processes behave like

in R except p_ℓ, which behaves exactly like in R until time $\tau + 1$, and crashes at that time. Since at most t processes crash in R', by definition of $\mathcal{A}$, eventually a leader must be elected. In fact, in R' all the processes except p_ℓ behave exactly like in R and elect p_ℓ as their (permanent) leader. These processes cannot distinguish R' from R and cannot detect the crash of p_ℓ. Hence, in R' protocol $\mathcal{A}$ does not satisfy the Eventual Leadership property of Ω, which is a contradiction. Therefore, p_ℓ cannot stop writing in the shared memory. □

Lemma 8 *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.*

Proof Assume, by way of contradiction, that a correct process p_i stops reading from the shared memory in run R at time τ. Let τ' be the time at which p_i chooses permanently p_ℓ as leader. Consider another run R' of $\mathcal{A}$ in which p_ℓ behaves exactly like in R until time $\max(\tau, \tau') + 1$, and crashes at that time. Since at most t processes crash in R', by definition of $\mathcal{A}$, a leader must be eventually elected. In R', we make p_i to behave exactly like in R. As it stopped reading the shared memory at time τ, p_i cannot distinguish R' from R and cannot detect the crash of p_ℓ. Hence in R', p_i elects p_ℓ as its (permanent) leader at time τ'. Hence, in R' protocol $\mathcal{A}$ does not satisfy the Eventual Leadership property of Ω, which is a contradiction. Therefore, p_i cannot stop reading from the shared memory. □

The following theorem follows immediately from the previous lemmas.

Theorem 3 *The protocol described in Fig. 2 is optimal 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 protocol described in Fig. 2 requires that each process (including the leader) reads forever the shared memory (all the processes have to read the array $SUSPICIONS[1..n, 1..n]$).

4 A t-Resilient Protocol for $\mathcal{AS}_{n,t}[AWB]$ with Bounded Variables Only

4.1 A Lower Bound Result

This section shows that any protocol that implements an eventual leader service Ω in $\mathcal{AS}_{n,t}[AWB]$ with only bounded memory has runs in which $t + 1$ correct processes have to read and write forever the shared memory. As we will see, it follows from this lower bound that the protocol described in Fig. 4 is optimal with respect to this criterion.

Let $\mathcal{A}$ be a protocol that implements Ω in $\mathcal{AS}_{n,t}[AWB]$ 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 a time after which no new memory positions are used, and each memory position has bounded number of bits.

Theorem 4 *The protocol $\mathcal{A}$ has runs in which at least $t+1$ processes write forever in the shared memory.*

Proof The intuition that underlies this theorem and its proof is the following. If no more than t processes write forever, it is not possible to distinguish between these processes having crashed or being very slow. Then, we need at least one more process that writes in order to be able to eventually elect a common leader.

To prove the claim we construct a run R of $\mathcal{A}$ such that:

1. R is fault free,
2. Process p_1 is synchronous while the rest of processes are asynchronous, and
3. There is an infinite sequence of times $\tau_0 < \tau_1 < \tau_2 < \ldots$ such that, $\forall i > 0$, in the interval $(\tau_{i-1}, \tau_i]$ some process changes its leader or at least $t+1$ processes write in the shared memory.

Clearly, since a leader must be eventually elected in R and the number of processes is finite, due to Item 3, there is a set of at least $t+1$ processes that write in the shared memory forever.

For simplicity, let us define $\tau_0 = 0$. This will be the base case. Then, for $i > 0$ let us assume R is already constructed up to time τ_{i-1}. We construct now interval $(\tau_{i-1}, \tau_i]$. This interval is constructed differently depending on which of the following two cases occurs.

- If at time τ_{i-1} the leader of some process p_j is an asynchronous process p_k (i.e., $k \neq 1$), we first consider a run R_i that behaves exactly like R up to time τ_{i-1}. Then, after that time all processes advance synchronously (e.g., one step per time unit), except p_k which crashes at time $\tau_{i-1}+1$. By Eventual Leadership, there is a time $\tau > \tau_{i-1}$ in R_i at which no process considers p_k as its leader. Then, let us define $\tau_i = \tau + 1$ and make R to behave in the interval $(\tau_{i-1}, \tau_i]$ as follows. All processes except p_k behave in this interval exactly like in the interval $(\tau_{i-1}, \tau_i]$ of R_i. Process p_k does not crash, but is stopped at time $\tau_{i-1}+1$ and does not execute any step until the end of the interval. This behavior is possible since p_k is asynchronous. Then, we have that in the interval $(\tau_{i-1}, \tau_i]$ some process changed its leader. This ends the first case.
- The second case occurs when at time τ_{i-1} in R the leader of all processes is the synchronous process p_1. As before we now consider an auxiliary run R_i that behaves exactly like R up to time τ_{i-1}. After that time all processes advance synchronously (e.g., one step per time unit) in R_i. If some process p_j changes its leader in R_i at some time $\tau > \tau_{i-1}$, then we define $\tau_i = \tau + 1$ and make the interval $(\tau_{i-1}, \tau_i]$ of R behave exactly as interval $(\tau_{i-1}, \tau_i]$ of R_i.

  Otherwise, if no process changes its leader in R_i after τ_{i-1}, we have from Lemma 7 that p_1 writes in the shared memory forever. Let us assume by way of contradiction that there is a time $\tau > \tau_{i-1}$ after which at most $t-1$ other processes write forever in the shared memory in R_i. Since the shared memory is bounded, some state (understood as the value of all its bits) S of the shared memory must occur infinitely often in R_i after τ. (First line in Fig. 3 where the state S is represented by an area with stripes.)

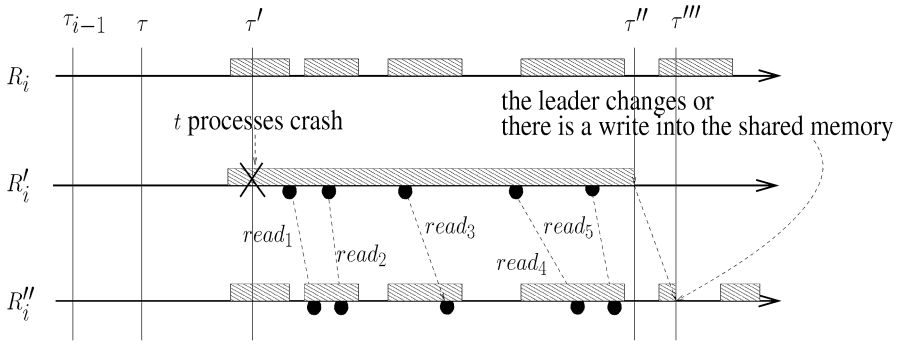

Fig. 3 Illustrating the runs R_i, R'_i and R''_i

Let us consider now a run R'_i which behaves exactly like R_i up to time $\tau' > \tau$ at which the shared memory is in state S (second line in Fig. 3). Then, at that time the (up to t) processes that were writing in the shared memory (including p_1) crash in R'_i. The rest of the processes advance synchronously (and hence the AWB_1 assumption holds in R'_i) until the smallest time $\tau'' > \tau'$ at which some process changes its leader or some process writes in the shared memory. This must eventually occur by Eventual Leadership, since the leader of all the processes at time τ' has crashed in R'_i. Note that in the interval (τ', τ'') all read operations find the shared memory in state S.

Consider now another run R''_i in which the up to t processes (including p_1) that write forever in R_i behave like they do in that run, while the rest of processes (let us denote this set of processes by L) behave like in R_i up to time τ' (last line in Fig. 3.) After τ', the processes in L are delayed (note that they are all asynchronous) so that every time they read form the shared memory they find it in state S (see Fig. 3). From the behavior of the processes in L in run R'_i and the fact that they cannot distinguish run R''_i from run R'_i, we have that there is a time $\tau''' > \tau'$ at which some process in L changes its leader or writes in the shared memory in run R''_i. Then, we define $\tau_i = \tau''' + 1$ and make interval $(\tau_{i-1}, \tau_i]$ of R behave exactly like that interval in R''_i.

Figure 3 summarizes the previous reasoning. In the first run R_i, after τ, only t processes write forever. The same state S (depicted by the area with stripes) occurs repeatedly forever. In the run R'_i, these t processes crash in state S (they crash at the time marked with a cross). The read operations from the other processes are indicated with black dots. In the run R''_i, the same processes as in R'_i read while the system in the state S. □

4.2 A Protocol with Only Bounded Variables

Principles and Description As already indicated, we are interested here in a protocol 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 1WMR 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 canceling it, namely, it resets $LAST[i, k]$ equal to $PROGRESS[i, k]$. So, p_i writes $PROGRESS[i, k]$ and $LAST[k, i]$, while p_k reads them. 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 2 below, this is the price that has to be paid to have bounded shared variables.

Using this simple technique, we obtain the protocol described in Fig. 4. Let us recall that p_i is the owner of $PROGRESS[i, k]$ and $LAST[k, i]$, $1 \leq k \leq n$, i.e., it is the only process that can write them. So, p_i manages two additional local arrays $progress_i[1..n]$ and $last_i[1..n]$, such that $progress_i[k]$ is a local copy of $PROGRESS[i, k]$, and $last_i[k]$ is a local copy of $LAST[k, i]$. (As in the first protocol, this allows saving shared memory accesses.)

In order to capture easily the parts that are new or modified with respect to the previous protocol, the line number of the new statements are suffixed with the letter R (so the line 11 of the previous protocol is replaced by six new lines 11.R1–11.R6, while each of the lines 20, 21 and 22 is replaced by a single line). This allows a better understanding of the common principles on which both protocols rely.

Proof of Correctness The statement of the Lemmas 1–6, and Theorem 1 are still valid when the shared array $PROGRESS[1..n]$ is 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 Lemma 1, Lemma 3, Lemma 4, Lemma 5, Lemma 6, and Theorem 1 are nearly verbatim the same.

The proofs of Lemma 2 has to be slightly modified to suit the new context. Basically, it differs from its counterparts of Sect. 3.4 in the way it establishes 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 $LAST[k, i]$".)

A reasoning similar to the one in the proof of Theorem 2 shows that each variable $SUSPICIONS[j, k]$, $1 \leq j, k \leq n$, is bounded. Combined with the fact that the variables $PROGRESS[j, k]$ and $LAST[j, k]$ are boolean, we obtain the following theorem.

Theorem 5 *All the variables used in the protocol described in Fig. 4 are bounded.*

Concerning the variables that are updated, we have the following theorem.

Theorem 6 *Let p_ℓ be the process elected as the eventual common leader in the protocol described in Fig. 4. There is a set of t processes p_i, $i \neq \ell$, such that eventu-*

task $T1$:
(1) **when** leader() **is invoked**:
(2) **for_each** $k \in \{1, ..., n\}$ **do**
(3) **let** $witness_i[k]$ = set of $(t+1)$ process identities such that
$\forall x \in witness_i[k],\ \forall y \notin witness_i[k]$: $(SUSPICIONS[x,k],x) < (SUSPICIONS[y,k],y)$;
(4) **let** $susp_i[k] = \Sigma_{x \in witness_i[k]}\ SUSPICIONS[x,k]$
(5) **end_for**;
(6) return(ℓ) **where** ℓ **is such that** $(-,\ell) = lex\_min(\{(susp_i[k],k)\}_{1 \le k \le n})$

task $T2$:
(7) **repeat_forever**
(8) **let** $my\_witnesses_i$ = set of $(t+1)$ processes identities such that
$\forall x \in my\_witnesses_i, \forall y \notin my\_witnesses_i$: $(SUSPICIONS[x,i],x)$
$< (SUSPICIONS[y,i],y)$;
(9) **let** $susp\_count_i = \Sigma_{x \in my\_witnesses_i}\ SUSPICIONS[x,i]$;
(10) **if** $\big((\text{leader}() = i) \vee (susp\_count_i \neq prev\_susp\_count_i)\big)$
(11.R1) **then for_each** $k \in \{1, \ldots, n\}$ **do**
(11.R2) $last\_k_i \leftarrow LAST[i,k]$;
(11.R3) **if** $=(progress_i[k] = last\_k_i)$
(11.R4) **then** $progress_i[k] \leftarrow \neg\ last\_k_i$; $PROGRESS[i,k] \leftarrow progress_i[k]$
(11.R5) **end_if**
(11.R6) **end_for**
(12) **end_if**;
(13) $prev\_susp\_count_i \leftarrow susp\_count_i$
(14) **end_repeat**

task $T3$:
(15) **when** $timer_i$ **expires**:
(16) $k \leftarrow$ leader();
(17) **let** $witness\_k_i$ = set of $(t+1)$ process identities such that
$\forall x \in witness\_k_i,\ \forall y \notin witness\_k_i$: $(SUSPICIONS[x,k],x) < (SUSPICIONS[y,k],y)$;
(18) **let** $susp\_k_i = \Sigma_{x \in witness\_k_i}\ SUSPICIONS[x,k]$;
(19) **if** $\big((k \neq i) \wedge (i \in witness\_k_i) \wedge (k = prev\_ld_i) \wedge (susp\_k_i = prev\_susp_i)\big)$
(20.R1) **then** $progress\_k_i \leftarrow PROGRESS[k,i]$;
(21.R1) **if** $(progress\_k_i \neq last_i[k])$
(22.R1) **then** $last_i[k] \leftarrow progress\_k_i$; $LAST[k,i] \leftarrow progress\_k_i$
(23) **else** $suspicions_i[k] \leftarrow suspicions_i[k] + 1$;
(24) $SUSPICIONS[i,k] \leftarrow suspicions_i[k]$
(25) **end_if**
(26) **end_if**;
(27) $prev\_ld_i \leftarrow k$; $prev\_susp_i \leftarrow susp\_k_i$;
(28) $timeout_i \leftarrow susp\_k_i$; set $timer_i$ to $timeout_i$

Fig. 4 t-resilient eventual leader election with all variables bounded (code for p_i)

ally 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 intuition of the proof is similar to that of Theorem 2, but in this case the hand-shaking mechanism forces the witnesses p_i of the leader p_ℓ to update their corresponding entries $LAST[\ell,i]$, and p_ℓ to update the corresponding entries $PROGRESS[\ell,i]$ (instead of $PROGRESS[\ell]$ as done in the algorithm in Fig. 2). The following is a detailed proof.

The proof that (1) there is a time after which the variables $SUSPICIONS[j,k]$, $1 \leq j,k \leq n$, are no longer written, and the proof that (2) there is a time after which $PROGRESS[x,j]$, $1 \leq x,j \leq n$, $x \neq \ell$, is no longer written, are the same as the proof done in Theorem 2. 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 $PROGRESS[x,y]$ (line 11.R3), it follows that there is a time after which the predicate $LAST[x,y] = PROGRESS[x,y]$ remains forever true for $1 \leq x,y \leq n$ and $x \neq \ell$. Consequently, after a finite time, the test of line 21.R1 is always false for p_x, $x \neq \ell$, and $LAST[x,y]$ is no longer written.

The fact that $SUSPICIONS[j,k]$, $1 \leq j,k \leq n$, eventually never changes implies that the set of witnesses of p_ℓ will eventually stabilize. A process p_x that is not in this set of *stable witnesses* of p_ℓ eventually stops writing $LAST[\ell,x]$, because the predicate at line 19 is always false. Once this has happened, p_ℓ will eventually set $PROGRESS[\ell,x] = \neg LAST[\ell,x]$ (line 11.R4). After that, the predicate at line 11.R3 remains forever false and $PROGRESS[\ell,x]$ is no longer written. Additionally, note that p_ℓ is always witness of itself, since initially $SUSPICIONS[\ell,\ell] = 0$, while $SUSPICIONS[x,\ell] = 1$ for all $x \neq \ell$, and $SUSPICIONS[\ell,\ell]$ never increases (from the $(k \neq i)$ sub-predicate at line 19). Note as well that $LAST[\ell,\ell]$ is never modified (also from the $(k \neq i)$ sub-predicate at line 19), and hence $PROGRESS[\ell,\ell]$ eventually stops being written.

Hence, only the variables $PROGRESS[\ell,x]$ and $LAST[\ell,x]$ for the set of t processes p_x, $x \neq \ell$, that are stable witnesses of p_ℓ are written forever. □

Finally, the next corollary follows directly from the above theorem and Theorem 4.

Corollary 2 *The protocol described in Fig. 4 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 timer notion (where the timer durations have to monotonically increase when the values to which they are set increase). This means that *AWB* is a particularly weak assumption.
- The second contribution is the design of two protocols that elect an eventual leader in any asynchronous shared memory system that satisfies the assumption *AWB*. In addition of being t-resilient (where t is the maximum number of processes allowed to crash), and being based only on one-writer/multi-readers atomic shared

variables, these protocols enjoy noteworthy properties. The first protocol 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 protocol uses (1) a bounded memory but (2) requires that $t + 1$ processes forever write the shared memory.

- The third contribution shows that the previous tradeoff (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 protocols are optimal, both 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 protocol. Is it possible to design a leader protocol 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 protocol optimal with respect to the size of the control information (bit arrays) it uses to have a bounded memory implementation? Finally, a very interesting question (suggested by a referee) is the following one: are there synchrony assumptions that allow designing an algorithm using only a linear (wrt the number of processes n) number of shared variables (as they use a matrix $SUSPICIONS[1..n, 1..n]$, the proposed algorithms require a quadratic number of shared variables)?

Acknowledgements We would like to thank the anonymous referees for their careful reading and valuable comments that helped us improve the content and the presentation of the paper.

References

1. Abraham, I., Chockler, G.V., Keidar, I., Malkhi, D.: Byzantine disk Paxos, optimal resilience with Byzantine shared memory. In: Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04), pp. 226–235. ACM Press, New York (2004)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: Proc. 22th ACM Symposium on Principles of Distributed Computing (PODC'03), pp. 306–314. ACM Press, New York (2003)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04), pp. 328–337. ACM Press, New York (2004)
4. Aguilera, M.K., Englert, B., Gafni, E.: On using network attached disks as shared memory. In: Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'03), pp. 315–324. ACM Press, New York (2003)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
7. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
8. Fernández, A., Jiménez, E., Raynal, M.: Electing an eventual leader in an asynchronous shared memory system. In: Proc. 37th International IEEE Conference on Dependable Systems and Networks (DSN'07), pp. 399–408. IEEE Computer Society Press, Los Alamitos (2007)
9. Gafni, E., Lamport, L.: Disk Paxos. Distrib. Comput. **16**(1), 1–20 (2003)
10. Gibson, G.A., Nagle, D., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., Zelenka, J.: A cost-effective high-bandwidth storage architecture. In: Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98), pp. 92–103. ACM Press, New York (1998)

11. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Proc. 20th Symposium on Distributed Computing (DISC'06). Lecture Notes in Computer Science, vol. 4167, pp. 376–390. Springer, Berlin (2006)
12. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Trans. Comput. **53**(4), 453–466 (2004)
13. Guerraoui, R., Raynal, M.: A leader election protocol for eventually synchronous shared memory systems. In: 4th International IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'06), pp. 75–80. IEEE Computer Society Press, Los Alamitos (2006)
14. Guerraoui, R., Raynal, M.: The alpha of asynchronous consensus. Comput. J. **50**(1), 53–67 (2007)
15. Herlihy, M.P.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **11**(1), 124–149 (1991)
16. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proc. 23th IEEE International Conference on Distributed Computing Systems (ICDCS'03), pp. 522–529. IEEE Computer Society Press, Los Alamitos (2003)
17. Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic sized data structure. In: Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'03), pp. 92–101. ACM Press, New York (2003)
18. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
19. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998) (the first version of Paxos appeared a DEC Tech Report in 1989)
20. Larrea, M., Fernández, A., Arévalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: Proc. 19th Symposium on Resilient Distributed Systems (SRDS'00), pp. 52–60. IEEE Computer Society Press, Los Alamitos (2000)
21. Lee, E.K., Thekkath, C.: Petal: distributed virtual disks. In: Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96), pp. 84–92. ACM Press, New York (1996)
22. Lo, W.-K., Hadzilacos, V.: Using failure detectors to solve consensus in asynchronous shared memory systems. In: Proc. 8th International Workshop on Distributed Computing (WDAG'94). Lecture Notes in Computer Science, vol. 857, pp. 280–295. Springer, Berlin (1994)
23. Malkhi, D., Oprea, F., Zhou, L.: Ω meets Paxos: leader election and stability without eventual timely links. In: Proc. 19th International Symposium on Distributed Computing (DISC'05). Lecture Notes in Computer Science, vol. 3724, pp. 199–213. Springer, Berlin (2005)
24. Mills, D.L.: Network Time Protocol (Version 3). Request for Comments (RFC) 1305, March 1992
25. Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: Proc. International IEEE Conference on Dependable Systems and Networks (DSN'03), pp. 351–360. IEEE Society Press, Los Alamitos (2003)
26. Mostefaoui, A., Raynal, M.: Leader-based consensus. Parallel Process. Lett. **11**(1), 95–107 (2001)
27. Mostéfaoui, A., Mourgaya, E., Raynal, M., Travers, C.: Time-free assumption to implement eventual leadership. Parallel Process. Lett. **16**(2), 189–208 (2006)
28. Mostéfaoui, A., Raynal, M., Travers, C.: Time-free and timeliness assumptions can be combined to get eventual leadership. IEEE Trans. Parallel Distrib. Syst. **17**(7), 656–666 (2006)
29. Powell, D.: Failure mode assumptions and assumption coverage. In: Proc. of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 386–395. IEEE Computer Society Press, Boston (1992)
30. Raynal, M.: A short introduction to failure detectors for asynchronous distributed systems. ACM SIGACT News, Distrib. Comput. Column **36**(1), 53–70 (2005)