A Leader Election Protocol for Eventually Synchronous Shared Memory Systems

Rachid Guerraoui, Michel Raynal

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N^{o} 1765

A LEADER ELECTION PROTOCOL FOR EVENTUALLY SYNCHRONOUS SHARED MEMORY SYSTEMS

RACHID GUERRAOUI MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

A Leader Election Protocol for Eventually Synchronous Shared Memory Systems

Rachid Guerraoui[*] Michel Raynal[**]

Systèmes communicants

Publication interne n ˚ 1765 — Novembre 2005 — 10 pages

Abstract: While protocols that elect an eventual common leader in asynchonous message-passing systems have been proposed, to our knowledge, no such protocol has been proposed for the shared memory communication model. This paper presents a leader election protocol suited to the shared memory model. In addition to its design simplicity, the proposed protocol has two noteworthy properties, namely, it does not use timers, and is optimal with respect to the number of processes that have to write forever the shared memory: a single process has to do it (namely, the leader that is eventually elected).

Among the many possible uses of such a leader protocol, one is Lamport's Paxos protocol. Paxos is an asynchronous consensus algorithm that relies on an underlying eventual leader abstraction. As recently, several versions of Paxos have been designed for asynchronous shared memory systems (the shared memory being an abstraction of a physically shared memory or a set of commodity disks that can be read and written by the processes), the proposed leader protocol makes Paxos effective in these systems.

Key-words: Asynchronous system, Commodity disk, Consensus, Efficiency, Eventual synchrony, Fault-tolerance, Leader service, Paxos, Shared memory system, Storage area network.

(Résumé : tsvp)

[*] Distributed Programming Lab, EPFL, Lausanne, Switzerland, rachid.guerraoui@epfl.ch

[**] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

Election d'un leader dans un système à mémoire partagé

Résumé : Ce rapport présente un protocole d'élection d'un leader dans un système à mémoire partagée inéluctablement synchrone.

Mots clés : Systèmes asynchrones, Tolérance aux fautes, Crash de processus, Leader inéluctable, Mémoire partagée.

1 Introduction

1.1 Motivation

Among the many different motivations that can direct system designers to provide processes with a leader service, we consider here only one of them, namely, Lamport's Paxos protocol [18]. This protocol (whose first version was developed for asynchronous message-passing systems) aims at constructing a replicated state machine on top of which one can be build highly available distributed services [20]. Since its first version, several papers have explained the basics Paxos relies on (e.g., [5, 15, 19]), or have formally proved its correctness (e.g., [9]). Paxos implementations can be found in [22, 33].

A component at the core of Paxos is its underlying "Synod algorithm" that is a consensus algorithm. Usually, when the word "Paxos" is used in the literature, the authors refer to this consensus component. We will do the same in this paper. Two noteworthy features characterizes Paxos.

- The first feature lies in its methodological decomposition. The Paxos consensus algorithm can be seen as a simple "addition" of two sub-protocols, one sub-protocol addressing the safety property of consensus (a single value is decided, and that value has been proposed by a process), the other sub-protocol addressing its liveness (all the non-faulty processes decide).
- The second noteworthy feature lies in the design of the sub-protocol that addresses consensus safety. That sub-protocol is based on news ideas and new techniques. Basically, it can be seen as building a one-shot storage object that, if accessed concurrently might not store anything, and if accessed sequentially stores the first deposited value and keeps it forever.

The second sub-protocol used by Paxos is to ensure that a value is eventually deposited in the one-shot storage object. This is realized by assuming that processes can be elected as leaders, and allowing only the leaders to compete to deposit a value in the one-shot storage object. To ensure termination, Paxos assumes that the underlying leader service eventually elects a single correct process to become the leader, thereby allowing a value to be eventually deposited in the one-shot object (if no process succeeds in depositing a value before). Then, as soon as a value has been deposited, the Paxos protocol can easily direct the processes to terminate (they all decide the deposited value).

So, Paxos assumes the existence of an underlying eventual leader service, but does not provide the corresponding sub-protocol. That is why it is said that Paxos is a *leader-based consensus protocol.* Other leader-based consensus protocols can be found in [14, 15, 27]. This means that these protocols work in an asynchronous system augmented with a *leader service* (also called *leader oracle* or *leader failure detector* in the literature [6, 32]). Following [7], this service is usually denoted Ω. It has been shown that Ω captures the minimal assumptions on failures that the processes have to be provided with in order asynchronous consensus can be solved [7]. "Minimal" means that the properties defining Ω are both necessary and sufficient.

While the design of a leader-based protocol assumes only a "black box" providing a leader service, the design of such a service requires that the underlying asynchronous system on top of which it is implemented satisfies additional assumptions.(The fact that no additional assumptions would be required would contradict the impossibility to solve consensus in a pure asynchronous system [11].)

1.2 Leader protocols in message-passing systems

The design of protocols building Ω in asynchronous message-passing systems has received a lot of attention. Basically, two approaches have been investigated. We briefly present them.

The timer-based approach The first approach, that we call *timer-based*, relies on the addition of timing assumptions [10]. Basically, this approach 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 an eventual leader facility in such "augmented" asynchronous systems are based on timeouts (e.g., [2, 3, 21]). They use successive approximations to eventually provide each process with an upper bound on transfer delays and processing speed. They differ mainly in the "quantity" of additional synchrony they consider, and in 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 f be an upper bound on the number of processes that may crash ($1 \leq f < n$, 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 $\Diamond f$-*source*. This means that p has f 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 as soon as the receiver has crashed). Let us notice that such a $\Diamond f$-*source* is not known in advance and can never be explicitly known. It is also shown in [2] that there is no leader protocol if the system has only $\Diamond(f-1)$-*source*s. A versatile adaptive timer-based approach has been developed in [24] for asynchronous systems in which a majority of processes never crash.

The message pattern-based approach The second approach (introduced in [25], and that we call *message pattern*-based approach) does not assume eventual bounds on process and communication delays. It considers that there is a correct process p and a set Q of f 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 $(n - f)$ 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 $\Diamond f$-*source* assumption. This approach has been applied to the construction of a leader protocol in [28], and extended to dynamic systems in [30] (a *dynamic system* being a system that the processes can dynamically enter and leave).

A *hybrid* protocol has been proposed in [26]. Hybrid means here that the protocol assumes that the channels are eventually timely or the message pattern is eventually satisfied, but it is not known in advance which assumption will be satisfied during a particular run of the protocol. The aim of this approach is to increase the assumption coverage, thereby improving fault-tolerance [31]. More generally, a very general protocol that combines the advantages of both of the previous approaches has been developed in [29].

1.3 Content of the paper and roadmap

This paper addresses the implementation of a leader service in an asynchronous shared-memory system. To our knowledge, to date, no such protocol has been proposed. The shared memory can be a physically shared memory or a network of attached disks (NAD) that constitutes a storage area network (SAN). As commodity disks are cheaper than computers, they become attractive for achieving fault-tolerance. This makes the leader protocol presented in the paper very relevant for such systems [1, 4, 13]. It could also be useful for real-time systems in which some components are based on asynchronous protocols.

The proposed protocol is based on the assumption that the underlying shared memory system is eventually synchronous. It enjoys two noteworthy properties. First, it is *timer-free*. Second, it is *write-optimal* in the sense that, after some time, a single process has to keep on writing the shared memory (optimality comes from the fact that at least the leader has to keep on writing forever the shared memory to inform the other processes that it has not crashed).

The paper consists in 5 sections. Section 2 defines the shared memory asynchronous system model and the leader election service Ω. Then Section 3 defines additional synchrony assumptions, presents a leader protocol based on these assumptions, and proves the protocol correctness (interestingly, in addition to the proof itself, this section helps also understanding the way the protocol works). Section 4 discusses the properties of the protocol and shows it is optimal in the sense that, after some finite time, a single process is required to keep on writing the shared memory. Finally, Section 5 concludes the paper.

2 System Model and Eventual Leader

2.1 Processes

The system consists in a finite set of $n > 1$ processes p_1, p_2, ..., p_n. Each process has an identity (id); the id of p_i is 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 operations as described by its protocol. After it has crashed, a process executes no more operation. By definition, a process is *faulty* during a run if it crashes during that run. Otherwise, it is *correct* during that run.

There is no assumption on the relative speed of a process with respect to another. We only assume that, until it possibly crashes, the speed of a process is positive (it cannot stop during an infinite period between two consecutive operations of its algorithm).

2.2 Shared Memory

The process communicate by writing and reading a memory made up of shared *registers*. Each register is *reliable*, 1W*R and *regular* [17].

Reliability means here that a register never crashes: it can always execute a read or a write operation issued by a process and never corrupts its value. If a process crashes while executing an operation, that operation is either fully executed, or not at all executed. 1W*R means that each register has a single writer (statically defined), but can be read by all the processes.

Finally, regularity defines the value returned by a read operation. Let R be a shared register.

- A read of R that is concurrent with no write of R returns the current value of R.
- A read of R that is concurrent with one or more write of R operations returns the value of R before these write operations or the value written by one of these operations.

It is important to see that regularity is weaker than atomicity. An atomic register R is such that each read or write operation appears to an external observer as if it has been executed instantaneously at some point of the time line, between its start event and its end event. The crucial difference between regularity and atomicity is the following. If two consecutive read operations $r1$ and $r2$ on the same register R (with $r1$ preceding $r2$) are concurrent with the same write operation w on R, it is possible that the first read operation $r1$ returns the value of R written by w (new value), while the second read operation $r2$ returns the value of R before the write operation w (old value). This is called a *new/old inversion*. Atomicity is regularity plus the prevention of new/old inversions [17].

A register can be seen as an abstraction that encapsulates a shared variable when we consider a physically shared memory, or a commodity disk when we consider network attached disks. The notion of storage area network has recently been used in several systems. The disks are directly attached to high speed networks accessible to the processes. A process can access raw disk data (mediated by disk controllers with limited CPU and memory capabilities). Versions of Paxos developed for network attached disks are described in [8, 12].

2.3 Eventual Leader Service

A *leader* oracle is a entity that provides each process with a function leader() that returns a process name each time it is invoked. A unique correct leader is eventually elected but there is no knowledge of when the leader is elected. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this "anarchy" period is over. The *leader* oracle (usually denoted Ω [7]) satisfies the following property:

- Validity: Any invocation of the primitive leader() that terminates returns a process id.
- Eventual Leadership[1]: There is a time t and a correct process p_i such that, after t, every invocation of leader() by any correct process returns i (the id of p_i).

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

- Termination: Any invocation of the primitive leader() issued by a correct process terminates.

The Ω leader abstraction has been introduced and formally developed in [7] 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., [14, 18, 27] for message-passing systems, and [12] for shared memory systems)[2].

3 Implementing a Leader Oracle

As already indicated, an eventual leader service cannot be implemented in a pure asynchronous (message-passing or shared memory) system [11]. As we have observed in the introduction, implementing such a service requires to "enrich" the underlying asynchronous system with additional "synchrony" properties. This section first states such properties suited to an asynchronous shared memory system. Then, it presents a protocol building a leader service in such an augmented asynchronous shared memory system, and proves it is correct.

3.1 Eventually Synchronous Shared Memory Systems

The shared memory system is assumed to satisfy the following additional property:

> [Eventually synchronous shared memory system] There is a time after which there are a positive lower bound and an upper bound for a process to execute a local step, a read or a write of a shared register.

It is important to notice that the values of the lower and upper bounds, and the time after which these values become the actual lower and upper bounds are not known. This additional assumption is the same as the partial synchrony assumption used in [6] to implement an eventually perfect failure detector in a message-passing system. As in [6, 10], the (finite but unknown) time after which the previous property is satisfied is called *global stabilization time* (GST).

3.2 An Eventual Leader Algorithm

Underlying principle The algorithm that, based on the previous assumption on the system behavior, build an eventual leader oracle is described in Figure 1. As in other leader protocols, the idea that underlies its design is for each process p_i to elect as the leader the process with the smallest id that it considers as being alive. As a process p_i never considers itself as crashed, at any time, the process it elects as its current leader has necessarily an id j such that $j \leq i$. The id of the process that p_i considers leader is locally stored in a local variable $leader_i$.

Shared memory The shared memory is composed of an array of n reliable 1W*R regular registers containing integer values. This array, denoted $PROGRESS[1..n]$, is initialized to $[0, \ldots, 0]$. Only p_i can write $PROGRESS[i]$. Any process can read any register $PROGRESS[j]$. The register $PROGRESS[i]$ is used by p_i to inform the other processes about its status.

Process behavior First, when a process p_i considers it is leader, it repeatedly increments its register $PROGRESS[i]$ in order to let the other processes know that it has not crashed (**while** loop and line 2).

Whether it is or not a leader, a process p_i increments a local variable l_clock_i (initialized to 0) at each step of the infinite **while** loop (line 3). This variable can be seen as a local clock that p_i uses to measure its local progress.

It is possible that p_i be very rapid and increments very often l_clock_i, while its current leader p_j is slow and two of its consecutive increments of $PROGRESS[j]$ are separated by a long period of time. This can direct p_i to suspect p_j to have crashed, and consequently to select another leader with a possibly greater

[2] 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.

```
when leader() is invoked by $p_i$: return $(leader_i)$

Background task $T$:
(1)  while $(true)$ do
(2)  if $(leader_i = i)$ then $PROGRESS[i] \leftarrow PROGRESS[i] + 1$ end_if;
(3)  $l\_clock_i \leftarrow l\_clock_i + 1$;
(4)  if $(l\_clock_i = next\_check_i)$ then
(5)     then $has\_ld_i \leftarrow false$;
(6)          for $j$ from $1$ to $(i-1)$ do
(7)               if $(PROGRESS[j] > last_i[j])$ then
(8)                  $last_i[j] \leftarrow PROGRESS[j]$;
(9)                  if $(leader_i \neq j)$ then $delay_i \leftarrow 2 \times delay_i$ end_if;
(10)                 $next\_check_i \leftarrow next\_check_i + delay_i$;
(11)                 $leader_i \leftarrow j$;
(12)                 $has\_ld_i \leftarrow true$;
(13)                 exit_for_loop
(14)              end_if
(15)         end_for;
(16)         if $(\neg has\_ld_i)$ then $leader_i \leftarrow i$ end_if
(17)    end_if
(18) end_while
```

Figure 1: Ω in an Eventually Synchronous Shared Memory System (code for p_i)

id. To prevent such a bad scenario from occurring, each process p_i handles another local variable denoted $next\_check_i$ (initialized to an arbitrary positive value, e.g., 1). This variable is used by p_i to compensate the possible drift between l_clock_i and $PROGRESS[j]$. More precisely, p_i tests if its leader has changed only when $l\_clock_i = next\_check_i$. Moreover, p_i increases the duration (denoted $delay_i$ and initialized to any positive value) between two consecutive checks (lines 9) when it discovers that its leader has changed. In all cases, it schedules the the logical date $next\_check_i$ at which it will check again for leadeship (line 10).

So, the core of its algorithm (lines 6-14), that consists for p_i in checking if its leader has changed and a new leader has to be defined, is executed only when $l\_clock_i = next\_check_i$. For doing this check, each p_i maintains a local array $last_i[1..(i-1)]$ such that $last_i[j]$ stores the last value of $PROGRESS[j]$ it has previously read (line 8). Moreover, when it tries to define its leader, p_i checks the processes always starting from p_1 until p_{i-1} (line 6). It stops at the first process p_j that did some progress since the last time p_i read $PROGRESS[j]$ (line 7). If there is such a process p_j, p_i considers it as its (possibly) new leader (line 11). If p_j was not its previous leader, p_i considers that it previously did a mistake and consequently increases the delay separating two checks for leadeship (line 9). In all cases, it then updates the logical date at which it will test again for leadership (increase of $next\_check_i$ at line 10). If, p_i sees no progress from any p_j such that $j < i$, it considers itself as the leader (line 16).

As indicated in the introduction, it is important to notice that the protocol is timer-free: no process is required to use a physical clock. The correctness of the protocol rests on a behavioral property of the underlying shared memory system (eventual synchrony), but does not need a special equipement (such as local physical clocks) to benefit from that eventual synchrony property.

3.3 Proof of the Protocol

The validity and termination properties defining the eventual leader service are easy and left to the reader. We focus here only on the proof of the eventual leadership property.

Theorem 1 *Let us assume that there is a time after which there are a lower bound and an upper bound for any process to execute a local step, a read or a write of a shared register. The algorithm described in Figure 1 eventually elects a single leader that is a correct process.*

Proof Let $t1$ be the time after with there are a lower bound and an upper bound on the time it take for a process to execute a local step, a read or a write of a shared register (global stabilization time). Moreover, let $t2$ be the time after which no more process crashes. Finally let $t = \max(t1, t2)$, and p_ℓ be the correct process with the smallest id. We show that, from some time after t, p_ℓ is elected by any process p_i.

Let us first observe that there is a time $t' > t$ after which no process p_k, such that $k < \ell$, competes with the other processes to be elected as a leader. This follows from the following observations:
- After t, p_k has crashed and consequently $PROGRESS[k]$ is no longer increased.
- After t, for each process p_i, there is a time after which the predicate $last_i[k] = PROGRESS[k]$ remains permanently satisfied, and consequently, p_i never executes the lines 8-13 with $j = k$, from which we conclude that p_k can no longer be elected as a leader by any process p_i.

It follows that after some time $t' > t$, as no process p_k $(k < \ell)$ increases its clock $PROGRESS[k]$, p_ℓ always exits the **for** loop (lines 6-15) with $has\_ld_\ell = false$, and considers itself as the permanent and definitive leader (line 16). Consequently, from t', p_ℓ increases $PROGRESS[\ell]$ each time it executes the **while** loop (lines 1-18).

We claim that there is a time after which, each time a process p_i executes the **for** loop (lines 6-15), we have $PROGRESS[\ell] > last_i[\ell]$ (i.e., p_i does not miss increases of $PROGRESS[\ell]$). It directly follows from this claim, line 11 (where $leader_i$ is now always set to ℓ), and the fact that all processes p_k such that $k < \ell$ have crashed, that p_i always considers p_ℓ as its leader, which proves the theorem.

Proof of the claim. To prove the claim, let us define two critical values. Both definitions consider durations after t', i.e., after the global stabilization time (so, both values are bounded).

- Let $\Delta_w(\ell)$ be the longest duration, after t', separating two increases of $PROGRESS[\ell]$.
- Let $\Delta_r(i, \ell)$ be the shortest duration, after t', separating two consecutive reading by p_i of $PROGRESS[\ell]$.

We have to show that, after some time and for any p_i, $\Delta_r(i, \ell) > \Delta_w(\ell)$ remains permanently true, i.e., we have to show that after some time the predicate $last_i[\ell] < PROGRESS[\ell]$ is true each time it is evaluated by p_i.

Let us first observe that, as p_ℓ continuously increases $PROGRESS[\ell]$, $last_i[\ell] < PROGRESS[\ell]$ is true infinitely often. If $last_i[\ell] < PROGRESS[\ell]$ is true while $leader_i \neq \ell$, p_i doubles the duration $delay_i$ (line 9) before which it will again check for a leader (line 4). This ensures that eventually we will have a time after which $\Delta_r(i, \ell) > \Delta_w(\ell)$ remains true forever. *End of the proof of the claim.* $\Box_{Theorem\ 1}$

4 Discussion

Write optimality In addition to its design simplicity, the proposed protocol has a noteworthy property related to efficiency, namely, it is communication-efficient. Communication efficiency has been defined in the context of message-passing systems [2]. That definition can easily be adapted to shared memory systems. In such a context, we say that a leader protocol is *write-optimal* if there is a finite time after which only one process keeps on writing the shared memory. Let us observe that this is the best that can be done as at least one process has to write forever the shared memory (if after some time the process that is the leader does not write the shared memory, there is no way for the other processes to know that it has not crashed).

During the "anarchy" period before the global stabilization time, it is possible that different processes have different leaders, and that each process has different leaders at different times. Theorem 1 has shown that such an anarchy period always terminates when the underlying shared memory system satisfies the "eventually synchronous" property.

To show that the algorithm is write-optimal, let us first observe that, each time a process p_j considers it is a leader, it increments its global clock $PROGRESS[j]$. It follows that when several processes consider they are leaders, several shared registers $PROGRESS[-]$ are increased. Interestingly, after the common correct leader has been elected, a single 1W*R register keeps on being increased. This means that a single shared register keeps growing, while the $(n-1)$ other shared registers stop growing. Consequently, the algorithm is communication-efficient. It follows that it is optimal with respect to this criterion (as at least one process has to continuously inform the others that it is alive).

Another synchrony assumption The reader can also check that the "eventual synchrony" assumption can be replaced by the following assumption: there is a time after which there is an upper bound τ on the ratio of the relative speed of any two non-crashed processes. Such a bound-based assumption can be seen as another way to place a limitation on the uncertainty created by the combined effect of asynchrony and failures. This type of additional assumption has been proposed in [23] and investigated in [16] to implement self-stabilizing failure detectors in message-passing systems.

5 Conclusion

This paper has presented a leader election protocol for asynchronous shared memory systems. To work, the protocol requires that the system eventually satisfies a synchrony assumption, namely, there is a time after which there are a lower bound and an upper bound for a process to execute a local step, a read or a write of the shared memory. The proposed protocol relies on particularly simple design principles. Moreover, it is timer-free and write-optimal (after some finite time, a single process keeps on writing the shared memory).

The proposed protocol is (to our knowledge) the first leader election protocol suited to a shared memory system. By providing a sub-protocol Paxos relies on, it allows a shared memory version of Paxos to work. In that sense, it allows Ω to meet Paxos in crash prone shared memory systems.

References

[1] Abraham I., Chockler G.V., Keidar I. and Malkhi D., Byzantine Disk Paxos, Optimal Resilience with Byzantine Shared Memory. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 226-235, 2004.

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

[3] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337, St. John's, Newfoundland (Canada), 2004.

[4] Aguilera M.K. and Gafni E., On Using Network Attached Disks as Shared Memory. *Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 315-324, 2003.

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

[6] Chandra T. and Toueg S., unreliable Failure Detectors for Resilient Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996. (First version: PODC 1991.)

[7] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996. (First version: PODC 1992.)

[8] Chockler G.V. and Malkhi D., Active Disk Paxos with Infinitely Many Processes. *Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'02)*, ACM Press, pp. 78-87, 2002.

[9] de Prisco R., Lampson B.W. and Lynch N.A., Revisiting the Paxos Algorithm. *Theoretical Computer Science*, 243(1-2):35-91, 2000.

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

[11] 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.

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

[13] 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.

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

[15] Guerraoui R. and Raynal M., The Alpha of Asynchronous Consensus. *Tech Report #1676*, IRISA, Université de Rennes 1 (France), 21 pages, 2005. http://www.irisa.fr/bibli/publi/pi/2005/1676/1676.html

[16] Hutle M. and Widder M., Time-Free Self-Stabilizing Local Failure Detection. *Tech Report 33/2004*, Department of Automation, Technische Universität Wien (Austria), 2004.

[17] Lamport L., On Interprocess Communication. *Distributed Computing*, 1(2):77-101, 1986.

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

[19] Lamport L., Paxos Made Simple. *ACM Sigact News, Distributed Computing Column*, 32(4):34-58, 2001.

[20] Lampson B.W., How to Build a Highly Available System Using Consensus. *Proc. 10th Int. Workshop on Distributed Algorithms (WDAG'96)*, Springer Verlag LNCS #1151, pp. 1-17, 1996.

[21] 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)*, IEEE Computer Society Press, pp. 52-60, Nüremberg (Germany), 2000.

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

[23] Le Lann G. and Schmid., How to Implement a Timer-Free Perfect Failure Detector in Partially Synchronous Systems. *Tech Report 183/1-127*, Department of Automation, Technische Universität Wien (Austria), 2003.

[24] 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.

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

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

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

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

[29] Mostéfaoui A., Raynal M. and Travers C., Time-free and Timeliness Assumptions can be Combined to Get Eventual Leadership. Submitted to publication. *Tech Report #1624*, IRISA, Université de Rennes 1 (France), 16 pages, 2005. http://www.irisa.fr/bibli/publi/pi/2004/1624/1624.html

[30] Mostefaoui A., Raynal M., Travers C., Patterson S., Agrawal A. and El Abbadi A., From Static Distributed Systems to Dynamic Systems. *Proc. 24th Int'l IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, IEEE Computer Society Press, Orlando (Florida), 2005.

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

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

[33] Thekkath C., Mann T. and Lee E.K., Frangipani: a Scalable Distributed File System. *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, ACM Press, pp. 224-237, 1997.