Fernández Anta A, Jiménez E, Raynal M. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(6): 1267–1281 Nov. 2010. DOI 10.1007/s11390-010-1100-9

Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony

Antonio Fernández Anta[1], *Senior Member, ACM, IEEE*, Ernesto Jiménez[2], and Michel Raynal[3]

[1]*Institute IMDEA Networks, Avenida del Mar Mediterraneo, 22, 28918 Leganés, Spain*

[2]*EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain*

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

E-mail: antonio.fernandez@imdea.org; ernes@eui.upm.es; raynal@irisa.fr

Received June 10, 2009; revised September 3, 2010.

Abstract This paper considers the eventual leader election problem in asynchronous message-passing systems where an arbitrary number t of processes can crash ($t < n$, where n is the total number of processes). It considers weak assumptions both on the initial knowledge of the processes and on the network behavior. More precisely, initially, a process knows only its identity and the fact that the process identities are different and totally ordered (it knows neither n nor t). Two eventual leader election protocols and a lower bound are presented. The first protocol assumes that a process also knows a lower bound α on the number of processes that do not crash. This protocol requires the following behavioral properties from the underlying network: the graph made up of the correct processes and fair lossy links is strongly connected, and there is a correct process connected to $(n - f) - \alpha$ other correct processes (where f is the actual number of crashes in the considered run) through eventually timely paths (paths made up of correct processes and eventually timely links). This protocol is not communication-efficient in the sense that each correct process has to send messages forever. The second protocol is communication-efficient: after some time, only the final common leader has to send messages forever. This protocol does not require the processes to know α, but requires stronger properties from the underlying network: each pair of correct processes has to be connected by fair lossy links (one in each direction), and there is a correct process whose $n - f - 1$ output links to the rest of correct processes have to be eventually timely. A matching lower bound result shows that any eventual leader election protocol must have runs with this number of eventually timely links, even if all processes know all the processes identities. In addition to being communication-efficient, the second protocol has another noteworthy efficiency property, namely, be the run finite or infinite, all the local variables and message fields have a finite domain in the run.

Keywords eventually timely and fair lossy links, eventual leader election, failure detector, omega leader oracle, process initial knowledge

1 Introduction

1.1 The Class of Eventual Leader Oracles *Ω*

Failure detectors[1-2] are at the core of many fault-tolerant protocols encountered in asynchronous distributed systems. Among them, the class of *Ω* failure detectors[3] is one of the most important. (This class is also called the class of leader oracles; when clear from the context, the notation *Ω* will be used to denote either the oracle/failure detector class or an oracle of that class.) Assuming that no two processes have the same identity (id), an *Ω* oracle provides each process with a read-only local variable that contains a process id. *Ω* is characterized by the fact that these local variables satisfy the following *eventual leadership property*: there is a finite time after which all the leader local variables contain the same id, which is the id of a correct process (a process that does not commit failures). So, *Ω* guarantees that a correct common leader is eventually elected, but there is no knowledge on when this common leader is elected. Let us observe that, before the finite (but unknown) time from which the common leader is elected, it is possible to have an arbitrary long anarchy period during which the leader of each process can change, processes have different leaders (possibly, some of them being crashed processes), etc. The reader interested in other types of failure detectors will find definitions and protocols implementing them in [1, 4-9].

Regular Paper

This work was partially supported by the Comunidad de Madrid under Grant No. S2009/TIC-1692, and the Spanish MEC under Grant Nos. TIN2007-67353-C02-01 and TIN2008-06735-C02-01.

A fundamental feature of the oracle class Ω lies in the fact that, among all the classes of oracles that provide processes with information on failures only, it is the weakest class that allows to solve the consensus problem[3]. This means that, given any run of a system prone to process crashes, Ω provides the weakest information, related to the failures occurring in that run, that is needed for the processes to be able solve the agreement problem. (This problem is at the core of the state machine replication paradigm.) Examples of leader-based consensus protocols can be found in [10-13][①]. In all these protocols, Ω is used to ensure the liveness property of the protocol. Example of consensus-based protocols that implement the state machine replication paradigm can be found in [1, 10, 14-15].

Unfortunately, Ω cannot be implemented in pure asynchronous distributed systems prone to process crashes. (Such an implementation would contradict the impossibility of solving consensus in such systems[16]. A direct proof of the impossibility to implement Ω in pure crash-prone asynchronous systems can be found in [17].) So, a main challenge of fault-tolerant distributed computing consists in identifying properties that are at the same time "weak enough" in order to be satisfied by "almost all" underlying systems, while being "strong enough" to allow to implement Ω in the runs in which they are satisfied.

1.2 Related Work: The *Timely Link* Approach to Implement Ω

The first implementations[1,3,18] of Ω in crash-prone asynchronous distributed systems considered a fully connected communication network where all links are reliable and eventually timely. A link is *eventually timely*[19] if there is a time τ_0 after which there is a (possibly unknown) bound δ such that, for any time $\tau \geqslant \tau_0$, a message sent at time τ is received by time $\tau + \delta$ (stronger versions of eventually timely links are defined and called partially synchronous in [1, 8, 20]). By convention, as soon as a process q has crashed, the link from any process p to q can be considered as behaving in a timely manner.

This approach has then been refined to obtain weaker assumptions. It has been shown in [19] that it is possible to implement Ω in a system where communication links are unidirectional, asynchronous and lossy, provided there is a correct process whose all output links are eventually timely. The corresponding protocol requires that all the correct processes send messages forever. It is also shown in [19] that, if there is additionally a correct process whose all input and output links are fair lossy[②], it is possible to design a *communication-efficient* Ω protocol (i.e., a protocol that guarantees that, after some time, only one process has to send messages forever). Let us observe that communication-efficiency, as introduced in [19], is a minimal condition. This is because, in order not to be falsely suspected to have crashed, at least the leader (or a witness of it) has to send messages forever.

The notion of *eventual t-source* has been introduced in [21] (in a system model where t denotes the maximal number of processes that can crash). An eventual t-source is a correct process that has t eventually timely output links. It is shown in that paper that this weak assumption is strong enough for implementing Ω. Two protocols based on an eventual t-source are presented in [21]. In addition to the eventual t-source, the first protocol (denoted ADFT1 in the following) requires only fair lossy links, but it is not communication-efficient (it demands each correct process to send messages forever). The second protocol (denoted ADFT2) is communication-efficient (after some time, only the leader sends messages forever), but this is obtained at an additional price, namely, each link has to be reliable and t output links of the eventual t-source are timely from the very beginning of the execution (i.e., it is a perpetual t-source).

A protocol building Ω when there is a process that eventually becomes forever t-accessible, and all other links are fair-lossy is described in [22]. The notion of *eventual t-accessibility* is orthogonal to the notion of eventually timely t-source in the sense that none of them encompasses the other one. More specifically, a process p is t-accessible at some time τ if there is a set Q of t processes such that a message broadcast by p at τ receives a response from each process of Q by time $\tau + \delta$ (where δ is a bound *known* by the processes). This notion requires $t < n/2$ (i.e., a majority of correct processes) to prevent process blocking. Its interest lies on the fact that the set Q of processes whose responses have to be received in a timely manner is not fixed and can be different at distinct times.

As mentioned above, the eventual t-source and eventual t-accessibility notions are incomparable. A new assumption that is weaker than both of them, namely *eventually moving t-source*, is proposed in [23]. An eventually moving t-source is a correct process such that, eventually, each message it sends is timely received by a set Q of t processes (a faulty process is

[①] It is important to notice that the first version of the Paxos algorithm[10], which uses a (transient) leader oracle, dates back to 1989, i.e., before the Ω formalism was introduced and investigated.

[②] Roughly speaking, a link is fair lossy if infinitely many messages are received when infinitely many messages are sent through the link.

assumed to always receive the messages timely), that can be different at distinct times. In [23] three Ω protocols based on an eventually moving t-source are presented. A lower bound of $\Omega(nt)$ on the communication complexity (links that carry messages forever) of any protocol based on an eventually moving t-source is also given. The second protocol refines the first protocol to achieve this bound. The third protocol refines the first one to have bounded timeouts.

A totally different approach to implement Ω is presented in [24]. That approach, that does not use timers, is based on a message-pattern assumption. It is shown in [25] that this approach and the timely link approach can be combined to obtain protocols with a greater assumption coverage[26].

1.3 Related Work: Implementing Ω When n Is Not Known

The election of an eventual leader in a system made up of n processes, in which each process knows only its identity and the fact that no two processes have the same identity, has recently been addressed in [27] where an Ω protocol is presented.

Let G_{ET}^{R} be the directed graph whose vertices are the correct processes in a run R (the processes that do not crash in the considered run), and where there is a directed edge from p to q iff the link connecting p to q is eventually timely. A directed path from p to q in G_{ET}^{R} is called an *eventually timely path*.

The protocol described in [27] elects a common correct leader, despite the fact that no process knows n, as soon as there is a correct process p such that, for any correct process $q \neq p$, there is an eventually timely path from p to q. This protocol (denoted as JAF in the following) is not communication-efficient.

1.4 Motivation and Content of the Paper

This paper is on the design of eventual leader election protocols in systems where a process knows neither n, nor t, nor the ids of the other processes (it knows only their domain, so, the context is the same as the one considered in [27]). It investigates behavioral assumptions on the links, that allow to implement Ω despite such a weak initial knowledge. From a theoretical side, this study provides us with a better understanding of the eventual leader election problem by enlarging the type of systems in which it can be solved. From a practical side, it provides protocols that can be used in applications such as sensor systems. In such systems, the number of sensors is bounded, but this number is usually not known by the sensors themselves, and no sensor knows initially the id of the other sensors that define the system. An example of use is when, to prevent interference and save energy, only one of the sensors has to eventually send data to a base station.

Two protocols and an impossibility result are presented[3]. In the following, "initial knowledge" refers the values of the constants (if any) used in the protocol, the type and the initial content of the local variables of each process, and the implicit assumptions common to all the processes (e.g., the fact that they are provided with the same code). Processes are synchronous in the sense that there are upper and lower bounds on their execution speeds.

First Protocol. The first protocol that is presented (denoted as FJR1 in the following) assumes that the initial knowledge of the processes is as follows.

(K1) A process knows initially neither n, nor t, nor the id of the other processes. (This means that a process cannot compute these values from the initial values of its local variables.) A process knows only its own id, the domain of the identities, the fact that the ids are totally ordered and how to order two given ids, and the fact that no two processes have the same identity. As an example, the value domain of the identities can be the set of integers, and the processes are initially assigned distinct integers.

(K2(α)) The processes know a lower bound (denoted α) on the number of correct processes. (This means that the assumption K2(α) is satisfied as soon as α processes do not crash.)

It is important to notice that α does not allow the processes to compute the values of n or t that define a particular system instance. Actually, α abstracts all the pairs (n,t) such that $n-t \geqslant \alpha$. The protocol FJR1 works for any such pair.

While it requires only the initial knowledge defined in K1 and K2(α), the protocol FJR1 is intended for the runs R where the underlying network satisfies the two following behavioral properties:

(C1) Each ordered pair of processes that are correct in R is connected by a directed path made up of correct processes and fair lossy links.

(C2) Given a process p correct in R, let $\mathit{reach}_R(p)$ be the set of the processes that are correct in R and accessible from p through directed paths made up of correct processes and eventually timely links. There is at least one correct process p such that $|\mathit{reach}_R(p)| \geqslant (n-f)-\alpha+1$, where f is the number of actual crashes during the run R. (Observe that p is included in $\mathit{reach}_R(p)$.)

[3] The assumptions or properties related to the initial knowledge of each process are identified by the letter K, while the ones related to the network behavior are identified by the letter C.

Observe that condition C2 imposes $\alpha \geqslant 1$. Otherwise, $(n-f)-\alpha+1$ is larger than the number of correct processes, $n-f$, and C2 cannot be satisfied. As we can see, in the runs where exactly α processes are correct (i.e., $\alpha = n - f$), C2 is trivially satisfied, which means that no link is then required to be eventually timely. In that sense (although their codes differ in many respects), FJR1 generalizes JAF. While JAF requires the existence of a correct process p with eventually timely paths to each other correct process, FJR1 reduces this number to $(n - f) - \alpha$. On the other hand, JAF does not need property C1.

An Impossibility Result. Then the paper presents an impossibility result that consists in an existential lower bound theorem. That theorem states that, in any system in the absence of any initial knowledge of the number of correct or faulty processes, there is no leader protocol that implements an eventual leader in all runs where f processes fail and less than $n - f - 1$ links eventually behave in a timely manner. This result holds even if the identities of all the processes are part of the initial knowledge.

Second Protocol. The paper then considers the design of a communication-efficient protocol when the process initial knowledge is restricted to (K1). This protocol (denoted FJR2) works in any run R that satisfies the following network behavioral properties:

(C1′) Each pair of correct processes is connected by (typed[4]) fair lossy links (one in each direction).

(C2′) There is a correct process whose output links to every correct process are eventually timely.

This protocol is communication-efficient (after some finite time, only the common leader sends messages forever). It also satisfies the following noteworthy property: be the execution finite or infinite, both the size of the local variables and the size of the messages remain finite. Differently from FRJ1, FRJ2 assumes that no link duplicates messages. Comparing FRJ2 with JAF, Property C1′ is not necessary in JAF, and Property C2′ is relaxed to eventually-timely paths instead of eventually-timely links. On the other hand, JAF is not communication efficient.

1.5 FJR1 and FJR2 with Respect to ADFT1 and ADFT2

This subsection briefly compares the cited protocols. First, the assumptions on the initial knowledge of n and t are weaker in FJR1 and FJR2 than in ADFT1 and ADFT2. However, ADFT1 can be rewritten without using the particular values of n and t as soon as the "differential" value of $n - t$ is known. As far as the behavioral properties on the links are concerned we have the following.

None of ADFT1 and FJR1 is communication-efficient. While ADFT1 is based on the existence of an eventual t-source and fair lossy links, FJR1 requires the existence of a correct process p connected through eventually timely paths to $(n - f) - \alpha$ other correct processes, and fair lossy paths connecting each pair of correct processes. Essentially, the requirements that ADFT1 imposes at the link level are relaxed to the path level in FJR1.

Both ADFT2 and FJR2 are communication-efficient. ADFT2 requires a t-source and reliable links, and the explicit use of n cannot be easily eliminated from it. Differently, FJR2 requires the existence of a correct process whose output links to the other correct processes are eventually timely, and fair lossy links between every pair of correct processes. Essentially, timely and reliable links in ADFT2 are relaxed into eventually-timely and fair-lossy links in FJR2.

It follows that, while the protocols ADFT1 and ADFT2 on one side, and FJR1 and FJR2 on the other side, investigate the same behavioral properties at the link level (fair lossy links and eventual timely links), they consider different global properties when we look at a more global level (defined by the processes and the underlying network). So, these protocols differ not only in their requirements on the initial knowledge of the processes, but also in the way they combine properties on individual links to obtain global behavioral properties that allow to implement an eventual leader oracle.

1.6 Organizatioin

The paper is made up of 6 sections. Section 2 presents the distributed system model. Section 3 presents FJR1 and proves its correctness. Section 4 states and proves the lower bound result. Section 5 presents FJR2 and proves its correctness. Finally, Section 6 concludes the paper.

2 Distributed System Model

2.1 Processes with Crash Failures

The system is made up of a finite set Π of n processes. A process is denoted p, q or p_i, where i is its index. The indexes are used for notational convenience. A process p_i has an identity id_i.

As indicated in the introduction, initially, a process p_i knows only the value domain of the ids, its own id, the fact that the ids are totally ordered, and how to

[4] In a typed fair lossy link messages have types, and the guarantees are defined for the messages of the same type.

order two given ids. This means that there is no way for a process to compute n, t, or the ids of other processes, from the initial values of its local variables[⑤]. Without loss of generality, we assume that for all $p_i, p_j \in \Pi$, $i < j$ iff $id_i < id_j$. Then, to simplify the exposition, we use in the following i instead of id_i as the identity of p_i.

A process can crash (stop executing). Once crashed, a process remains crashed forever. A process executes correctly until it possibly crashes. A process that crashes in a run is *faulty* in that run, otherwise it is *correct*. The model parameter t denotes the maximum number of processes that can crash in a run ($1 \leqslant t < n$), while f denotes the number of actual crashes in a given run ($0 \leqslant f \leqslant t$). As for n, a process cannot determine the value of t from the values of its local variables. The code of the first protocol (FJR1) uses a constant α. This constant is assumed to be a lower bound on the number of correct processes.

As in [21], the processes are synchronous in the sense that there are lower and upper bounds on the number of processing steps they can execute per time unit. Each process has also a local clock that can accurately measure time intervals. The clocks of the processes are not synchronized.

2.2 Communication Network

The processes communicate by exchanging messages over links. Each pair of processes is connected by two directed links, one in each direction.

Communication Primitive. The processes are provided with a broadcast primitive that allows each process p to simultaneously send the same message m to the rest of processes in the system (e.g., like in Ethernet networks, radio networks, or IP-multicast). It is nevertheless possible, depending on the quality of the connectivity (link behavior) between p and each process, that the message m is received in a timely manner by some processes, asynchronously by other processes, and not at all by another set of processes.

Individual Link Behavior. A link cannot create or alter messages, but does not guarantee that messages are delivered in the order in which they are sent.

Concerning timeliness or loss properties, the communication system offers three types of links. Each type defines a particular quality of service that the corresponding links are assumed to provide.

- *Eventual Timely Link.* The link from p to q is *eventually timely* if there is a time τ_0 and a bound δ such that each message sent by p to q at any time $\tau \geqslant \tau_0$ is received by q by time $\tau + \delta$, if q is correct (τ and δ are not a priori known and can never be known). If process q is faulty, we assume that as soon as q has crashed, the link from p to q is timely.
- *Fair Lossy Link.* Let us assume that each message has a type. The link from p to q is *fair lossy* if, for each type μ, assuming that p sends to q infinitely many messages of the type μ, q (if it is correct) receives infinitely many messages of type μ from p.
- *Lossy Link.* The link from p to q is *lossy* if it can lose an arbitrary number of messages (possibly all the messages it has to carry).

As we can see, fair lossy links and lossy links are inherently asynchronous, in the sense that they guarantee no bound on message transfer delays. An eventually timely link can be asynchronous for an arbitrary but finite period of time. Concerning message duplication, a link satisfies property (D) if it is allowed to duplicate messages, and satisfies property (ND) if it is not allowed to.

Global Properties Related to the Communication System. R being a run, let G^R_{ET} be the directed graph whose vertices are the processes that are correct in R, and where there is a directed edge from p to q if the link from p to q is eventually timely in R. Similarly, let G^R_{FL} be the directed graph whose vertices are the correct processes, and where there is a directed edge from p to q if the link from p to q is fair lossy. (Notice that G^R_{ET} is a subgraph of G^R_{FL}.) Given a correct process p, $reach_R(p)$ has been defined in the introduction. It is the subset of correct processes q that can be reached from p in the graph G^R_{ET} (i.e., there is a path made up of eventually timely links and correct processes from p to each $q \in reach_R(p)$.)

As already indicated in the introduction, given an arbitrary run R, we consider the following behavioral properties on the communication system:

(C1) The graph G^R_{FL} is strongly connected.

(C1′) Each pair of correct processes is connected by fair lossy links (one in each direction).

(C2) There is (at least) one correct process p such that $|reach_R(p)| \geqslant (n - f) - \alpha + 1$. Recall that α is a lower bound on the number of correct processes.

- (C2′): There is a correct process whose output links to every correct process are eventually timely.

As noticed in the introduction, the property (C2) is always satisfied in the runs where $\alpha = n - f$ (exactly α processes are correct). Moreover, (C1′) and (C2′) are stronger than (C1) and (C2), respectively.

2.3 The Class Ω of Oracles

Introduced in [3], the *leader* oracle Ω has been defined informally in the introduction. It is a distributed

[⑤] A similar type of assumption on the initial knowledge of the process ids is encountered in the adaptive renaming problem[28-29].

entity that provides each processes p_i with a read-only local variable LEADER_i that contains a process id. When taken collectively, these local variables satisfy the following property:

Eventual Leadership. There is a finite time τ such that, after τ, the local variables LEADER_i of all the correct processes p_i contain forever the same identity, that is the identity of a correct process.

3 A Leader Election Protocol

Assuming that each process knows its identity (K1), the lower bound α on the number of correct processes (K2(α)), and that all the processes have distinct and comparable identities, the protocol FJR1 described in this section elects a leader in any run where the underlying communication network satisfies the properties (D), (C1) and (C2). Moreover, as far as the definition of *fair lossy link* is concerned, all the messages sent by the processes have the same type.

3.1 Description of the Protocol

As in other leader protocols, the underlying principle is as follows: a process elects as its current leader the process that it considers alive and it perceives as the "least suspected". The notion of "suspected" is implemented with counters, and "less suspected" means "smallest counter" (using process ids to tie-break equal counters). The protocol is described in Fig.1. It is composed of two tasks. To guarantee correctness, it is assumed that the sequences of statements defined by the Lines 02∼09 (body of the **repeat** loop) in task $T1$, the Line 11 of task $T2$, and the Lines 12∼25 of task $T2$ are executed in mutual exclusion.

Let X be a set of pairs ⟨counter, process id⟩. The function *lex_min*(X) returns the smallest pair in X according to lexicographical order.

Local Variables. The local variables shared and managed by the two tasks are the following ones.

• LEADER_i is the local variable the protocol has to assign values. It is initialized to i, and contains the id of the current leader of p_i.

• $members_i$ is a set containing all the process ids that p_i is aware of.

• $timer_i[j]$ is a timer used by p_i to check if the link from p_j is timely. The current value of $timeout_i[j]$ is used as the corresponding timeout value; it is increased each time $timer_i[j]$ expires.

```
 Init: allocate $susp\_level_i[i]$ and $suspected\_by_i[i]$; $susp\_level_i[i] \leftarrow 0$; $suspected\_by_i[i] \leftarrow \varnothing$; $members_i \leftarrow \{i\}$; $to\_reset_i \leftarrow \varnothing$;
       $silent_i \leftarrow \varnothing$; $sn_i \leftarrow 0$; $\text{LEADER}_i \leftarrow i$; $state_i \leftarrow \{(i, sn_i, \{(susp\_level_i[i], i)\}, silent_i)\}$   % initial knowledge (K1) %
Task $T1$:
(01)  repeat forever every $\eta$ time units
(02)      $sn_i \leftarrow sn_i + 1$;
(03)      for each $j \in silent_i$ do $suspected\_by_i[j] \leftarrow suspected\_by_i[j] \cup \{i\}$ end for;
(04)      for each $j \in members_i$ such that $|suspected\_by_i[j]| \geqslant \alpha$ do % initial knowledge (K2($\alpha$)) %
(05)                    $susp\_level_i[j] \leftarrow susp\_level_i[j] + 1$; $suspected\_by_i[j] \leftarrow \varnothing$ end for;
(06)      replace $(i, -, -, -)$ in $state_i$ by $(i, sn_i, \{(susp\_level_i[j], j) \mid j \in members_i\}, silent_i)$;
(07)      broadcast ($state_i$);
(08)      for each $j \in to\_reset_i$ do set $timer_i[j]$ to $timeout_i[j]$ end for; $to\_reset_i \leftarrow \varnothing$
(09)      $\text{LEADER}_i \leftarrow \ell$ such that $(-, \ell) = lex\_min(\{(susp\_level_i[j], j)\}_{j \in members_i})$
(10)  end repeat
Task $T2$:
when $timer_i[j]$ expires:
(11)      $timeout_i[j] \leftarrow timeout_i[j] + 1$; $silent_i \leftarrow silent_i \cup \{j\}$
when $state\_msg$ is received:
(12)  let $K = \{ (k, sn\_k, cand\_k, silent\_k) \mid$
          $(k, sn\_k, cand\_k, silent\_k) \in state\_msg \wedge \nexists (k, sn', -, -) \in state_i$ with $sn' \geqslant sn\_k \}$;
(13)  for each $(k, sn\_k, cand\_k, silent\_k) \in K$ do
(14)      if $k \in members_i$ then replace $(k, -, -, -)$ in $state_i$ by $(k, sn\_k, cand\_k, silent\_k)$;
(15)                    stop $timer_i[k]$; $to\_reset_i \leftarrow to\_reset_i \cup \{k\}$; $silent_i \leftarrow silent_i \setminus \{k\}$
(16)                else add $(k, sn\_k, cand\_k, silent\_k)$ to $state_i$;
(17)                    allocate $susp\_level_i[k]$, $suspected\_by_i[k]$, $timeout_i[k]$ and $timer_i[k]$;
(18)                    $susp\_level_i[k] \leftarrow 0$; $suspected\_by_i[k] \leftarrow \varnothing$; $timeout_i[k] \leftarrow \eta$;
(19)                    $members_i \leftarrow members_i \cup \{k\}$; $to\_reset_i \leftarrow to\_reset_i \cup \{k\}$
(20)      end if
(21)  end for;
(22)  for each $(k, sn\_k, cand\_k, silent\_k) \in K$ do
(23)      for each $(sl, \ell) \in cand\_k$ do $susp\_level_i[\ell] \leftarrow \max(susp\_level_i[\ell], sl)$ end for;
(24)      for each $\ell \in silent\_k$ do $suspected\_by_i[\ell] \leftarrow suspected\_by_i[\ell] \cup \{k\}$ end for
(25)  end for
```

Fig.1. Eventual leader protocol FJR1 (code for p_i).

- $silent_i$ is a set containing the ids j of all the processes p_j such that $timer_i[j]$ has expired since its last resetting.
- to_reset_i is a set containing the ids k of the processes p_k whose timer has to be reset.
- $susp\_level_i[j]$ contains the integer that locally measures the current suspicion level of p_j. It is the counter used by p_i to determine its current leader (see the update of LEADER$_i$ in Line 09 of Task $T1$).
- $suspected\_by_i[j]$ is a set used by p_i to manage the increases of $susp\_level_i[j]$. Each time p_i knows that a process p_k suspects p_j it includes k in $suspected\_by_i[j]$. Then, when the number of processes in $suspected\_by_i[j]$ reaches the threshold α, p_i increases $susp\_level_i[j]$ and resets $suspected\_by_i[j]$ to $\varnothing$ for a new observation period.
- sn_i is a local counter generating the sequence numbers attached to each message sent by p_i.
- $state_i$ is a set containing an element for each process p_k that belongs to $members_i$, namely, the most recent information issued by p_k that p_i has received so far (directly from p_k or indirectly from a path involving other processes). That information is a quadruple $(k, sn_k, cand_k, silent_k)$ where the component $cand_k$ is the set $\{(susp\_level_k[\ell], \ell) \mid \ell \in members_k\}$ from which p_k elects its leader.

Process Behavior. The aim of the first task of the protocol is to disseminate to all the processes the latest state known by p_i. That task is made up of an infinite loop (executed every η time units) during which p_i first updates its local variables $suspected\_by_i[j]$ and $susp\_level_i[j]$ according to the current values of the sets $silent_i$ and $members_i$. Then p_i updates its own quadruple in $state_i$ to its most recent value (which it has just computed) and broadcasts it (this is the only place of the protocol where a process sends messages). Finally, p_i resets the timers that have to be reset, updates accordingly to_reset_i to $\varnothing$, and recomputes the value of the LEADER$_i$ local variable it is implementing.

The second task is devoted to the management of the timer expiration and message reception. The code associated with the two first lines of this task is self-explanatory. When it receives a message (denoted $state\_msg$), a process p_i considers and processes only the quadruples that provide it with new information, i.e., the quadruples $(k, sn\_k, cand\_k, silent\_k)$ such that it has not yet processed a quadruple $(k, sn', -, -)$ with $sn' \geqslant sn\_k$. For each such quadruple, p_i updates $state_i$ (it also allocates new local variables if k is the id of a process it has never heard of before). Finally, p_i updates its local variables $susp\_level_i[\ell]$ and $suspected\_by_i[\ell]$ according to the information it learns from each new quadruple $(k, sn\_k, cand\_k, silent\_k)$ it has received in $state\_msg$.

3.2 Proof of the Protocol

Considering that each processing block (body of the loop in Task $T1$, timer expiration and message reception managed in Task $T2$) is executed atomically, we have $(j \in members_i)$ iff $((j, -, -, -) \in state_i)$ iff ($suspected\_by_i[j]$ and $susp\_level_i[j]$ are allocated). We also have ($timer_i[j]$ and $timeout_i[j]$ are allocated) iff $(j \in members_i \setminus \{i\})$. It follows from these observations that all the local variables are well-defined: they are associated exactly with the processes known by p_i. Moreover, a process p_i never suspects itself, i.e., we never have $i \in silent_i$ (this follows from the fact that, as $timer_i[i]$ does not exist, that timer cannot expire — the timer expiration in $T2$ is the only place where a process id is added to $silent_i$, Line 11 of Fig.1).

The proof considers an arbitrary run R.

Lemma 1. *Let $(k, sn, -, -)$ be a quadruple received by a correct process p_i. All the correct processes eventually receive a quadruple $(k, sn', -, -)$ such that $sn' \geqslant sn$.*

Proof. To prove the lemma, let us consider the first correct process p_i that receives a quadruple $(k, sn, -, -)$ such that no quadruple $(k, sn', -, -)$ with $sn' \geqslant sn$ belongs to $state_i$.

If $state_i$ does not contain a quadruple $(k, -, -, -)$, p_i adds $(k, sn, -, -)$ to $state_i$ (Line 16). Otherwise, p_i replaces in $state_i$ the old quadruple $(k, -, -, -)$, by the new one $(k, sn, -, -)$ (Line 14). Then, the only reason for the quadruple $(k, sn, -, -)$ to disappear from $state_i$, is its replacement by a quadruple $(k, sn'', -, -)$ such that $sn'' > sn$ (Lines 12 and 14). As 1) all the correct processes p_j broadcast regularly their current value of $state_j$ to all the processes, and 2) the graph $G^R_{\rm FL}$ is strongly connected, it follows that each correct process eventually receives the quadruple $(k, sn', -, -)$ or a quadruple $(k, sn'', -, -)$ with $sn'' > sn'$. □

Let L be the set that contains all the processes p_i that are correct in R and are such that $|reach_R(i)| \geqslant (n - f) - \alpha + 1$. By property (C2), we have $L \neq \varnothing$.

Lemma 2. *Let p_i be a process in L. There is a time after which, for any process p_j in $reach_R(i)$, $i \in silent_j$ remains permanently false.*

Proof. Let us first observe that i never belongs to $silent_i$. Now, for every $j \neq i$, observe that, in order to include i into $silent_j$, $timer_j[i]$ has to expire (Line 11). So, to prove the lemma, we show that, for every $j \in reach_R(i)$, $j \neq i$, there is a time at which $i \notin silent_j$ and after which $timer_j[i]$ never expires.

As p_i is correct, it periodically issues $state_i$ messages (Lines 06~07), each containing a quadruple $(i, sn_i, -, -)$. As $G^R_{\rm FL}$ is strongly connected, it follows that there is a time after which each process p_j such that $j \in \mathit{reach}_R(i)$ (let us remind that G^R_{ET} is a subgraph of G^R_{FL}) receives a message carrying one such quadruple (the quadruple progressed through paths in G^R_{FL}) and consequently allocates two local variables $\mathit{timer}_j[i]$ and $\mathit{timeout}_j[i]$ (Line 17).

The consecutive quadruples $(i, \mathit{sn}_i, -, -)$ periodically sent by p_i within state_i messages (Lines 06∼07) contain strictly increasing sequence numbers (Line 02). It follows that any p_j such that $j \in \mathit{reach}_R(i)$ receives (through the paths in G^R_{FL}) an infinite number of messages carrying quadruples $(i, \mathit{sn}_i, -, -)$, and that infinitely often these arrive when $(i, \mathit{sn}', -, -) \in \mathit{state}_j$ with $\mathit{sn}_i > \mathit{sn}'$. It follows that infinitely often p_j executes $\mathit{silent}_j \leftarrow \mathit{silent}_j \setminus \{i\}$ (Line 15).

The rest of the proof consists in showing that there is a time after which $\mathit{timer}_j[i]$ never expires. Due to the very definition of the graph G^R_{ET}, there is a time τ_0 after which all the links of this graph guarantee a (possibly unknown) bounded transfer delay δ. Let $\Delta^{\tau_0}_{i,j}$ be the value of $\mathit{timeout}_j[i]$ at time τ_0, and d the distance from p_i to p_j in G^R_{ET} $(1 \leqslant d \leqslant n-1)$.

Case 1. If $\Delta^{\tau_0}_{i,j} > d(\eta+\delta)$, due to the very definition of G^R_{ET}, $\mathit{timer}_j[i]$ can no longer expire because it is regularly stopped at Line 15 before $\Delta^{\tau_0}_{i,j}$ time units have elapsed since its last resetting at Line 08 (let us notice that different quadruples $(i, -, -, -)$ can take distinct paths in G^R_{ET} from p_i to p_j).

Case 2. If $\Delta^{\tau_0}_{i,j} \leqslant d(\eta+\delta)$, it is possible that $\mathit{timer}_j[i]$ expires before a quadruple $(i, \mathit{sn}_i, -, -)$ with $\mathit{sn}_i > \mathit{sn}'$ (where $(i, \mathit{sn}', -, -) \in \mathit{state}_j$) arrives at p_j. That process consequently increases $\mathit{timeout}_j[i]$ (Line 11). But this can happen only a finite number of times (namely, $d(\eta+\delta) - \Delta^{\tau_0}_{i,j} + 1$ times), after which we are in Case 1. □

Lemma 3. *Let p_i be a process in L. There is a time after which the local variables $\mathit{susp\_level}_k[i]$ of all the correct processes p_k remain forever equal to the same bounded value* (*denoted as SL_i*).

Proof. Let us first observe that any local variable $\mathit{susp\_level}_k[\ell]$ can be updated only at Line 05 or Line 23, and can only increase. p_k being any process, let us examine its local variable $\mathit{susp\_level}_k[i]$ where i is such that p_i belongs to L (i.e., p_i is a correct process such that $|\mathit{reach}_R(i)| \geqslant (n-f)-\alpha+1$). Due to Lemma 2, there is a time after which there is a set of at least $|\mathit{reach}_R(i)|$ correct processes p_j whose local predicate $i \in \mathit{silent}_j$ remains false forever. Moreover, there is a time after which the f faulty processes have crashed (before crashing, they sent a finite number of messages, and, after that time, they no longer send messages).

It follows from these observations that there is a finite time τ after which at most β processes p_ℓ can send state_ℓ messages including $(\ell, -, -, \mathit{silent}_\ell)$ with $i \in \mathit{silent}_\ell$. These β processes can be all the processes but the $(n-f)-\alpha+1$ processes of $\mathit{reach}_R(i)$ and the f faulty processes, i.e., $\beta \leqslant n-(n-f-\alpha+1)-f = \alpha-1$. It follows that after some finite time (when all the quadruples $(x, -, -, \mathit{silent}_x)$ with $i \in \mathit{silent}_x$ disseminated from the f faulty processes p_x have arrived or are lost), no process p_k can increase its local variable $\mathit{susp\_level}_k[i]$ at Line 05. By the gossiping of the state_k messages (Lemma 1), and the fact that the graph G^R_{FL} is strongly connected, it follows that the variables $\mathit{susp\_level}_k[i]$ of all the correct processes become equal (Line 23) and keep forever their common value (denoted as SL_i), which proves the lemma. □

Given a run, let B be the set of processes p_i such that $\mathit{susp\_level}_k[i]$ remains bounded at some correct process p_k.

Lemma 4. *The following holds*:

1) $B \neq \varnothing$.

2) *$\forall i \in B$, the local variables $\mathit{susp\_level}_k[i]$ of all the correct processes p_k remain forever equal to the same bounded value* (*denoted as SL_i*).

Proof. $L \subseteq B$ directly follows from the definition of B and Lemma 3. As $L \neq \varnothing$, we have $B \neq \varnothing$.

Let p_i be a process in B. The fact that the local variables $\mathit{susp\_level}_k[i]$ of all the correct processes p_k remain forever equal is an immediate consequence of Line 23 and the gossiping mechanism used to propagate the quadruples $(i, \mathit{sn}_i, -, -)$ (Lemma 1). □

Lemma 5. *For every faulty process p_i, either at each correct process p_j always $i \notin \mathit{members}_j$, or at each correct process p_j, $\mathit{susp\_level}_j[i]$ increases without bound.*

Proof. If no correct process p_j ever receives a message including a quadruple $(i, \mathit{sn}_i, -, -)$, then the variable $\mathit{members}_j$ of all the correct processes trivially remain such that $i \notin \mathit{members}_j$.

So, let us consider the case where at least one correct process p_k receives a quadruple $(i, \mathit{sn}_i, -, -)$ initially sent by p_i. Moreover, let us consider the last such quadruple received by any correct process. Due to the gossiping of the last quadruple $(i, \mathit{sn}_i, -, -)$ received (Lemma 1), and the fact that the graph G^R_{FL} is strongly connected, it follows that all the correct processes receive and process this quadruple. Then, they receive no message carrying a quadruple $(i, \mathit{sn}'_i, -, -)$ with $\mathit{sn}'_i > \mathit{sn}_i$ (this follows from the definition of $(i, \mathit{sn}_i, -, -)$ that is the last quadruple sent by p_i). Each correct process p_k then sets $\mathit{timer}_k[i]$ to $\mathit{timeout}_k[i]$. As no quadruple $(i, \mathit{sn}'_i, -, -)$ with $\mathit{sn}'_i > \mathit{sn}_i$ is ever received, it follows that 1) $\mathit{timer}_k[i]$ expires and p_k adds p_i to silent_k (at Line 11); and 2) i is never withdrawn from silent_k (at Line 15). It follows that each *state_msg* message (of the infinite sequence of such messages sent by p_k) carries a quadruple $(k, -, -, silent_k)$ with $i \in silent_k$.

It follows from the previous discussion, the fact that there are at least α correct processes, and the fact that after some finite time each correct p_k always suspects p_i (i.e., after some time i remains forever in $silent_k$), that, at each correct process p_j, $|suspected\_by_j[i]|$ becomes $\geqslant \alpha$ (at Line 24) infinitely often. Consequently, each correct process p_j infinitely often increases $susp\_level_j[i]$ (Line 05) which proves the lemma. □

Theorem 1. *The protocol described in Fig.1 ensures that, after some finite time, all the correct processes have forever the same correct leader.*

Proof. Due to Lemma 4, eventually all the correct processes p_k are such that $B \subseteq members_k$. Moreover, due to Lemma 5, B contains only correct processes.

As after some time, for each $j \in B$, each correct process p_k keeps forever the same bounded value SL_j in $susp\_level_k[j]$ (Lemma 4), it follows that all the correct processes p_i eventually output the same process id each time they read local variable LEADER$_i$, and that id is the identity of a correct process. □

4 A Lower Bound

The previous protocol FJR1 is not communication-efficient (each correct process has to send messages forever). Several communication-efficient eventual leader protocols (e.g., [21]) have been designed for systems in which each process initially knows the whole set of identities. The next section presents a communication-efficient leader election protocol (FJR2) where the initial knowledge of each process is limited to its id only. This protocol is based on the network behavioral assumptions (C1′) and (C2′) that are stronger than (C1) and (C2). Before describing this protocol, this section shows an associated lower bound on the network behavior when processes have no other initial knowledge on the number of faulty processes than $t = n - 1$. The lower bound states that it is impossible to implement Ω if all runs have less than $n - f - 1$ eventually timely links, even if each process initially knows the whole set of identities. Hence, the assumption (C2′) is existentially optimal on the number of eventually timely links.

The following lemma exploits the fact that a process has a limited initial knowledge on the number of correct or faulty processes. However, processes may know the system membership. This lemma is then used as the cornerstone in the proof of the lower bound.

Lemma 6. *Let us consider a system in which processes have no other initial knowledge on the number of correct or faulty processes than $t = n - 1$. Let P be any protocol that implements an eventual leader in this system. If, in an infinite run of P, a correct process p stops receiving messages from the rest of processes at some time τ, it eventually considers itself as the leader at some time $\tau' \geqslant \tau$.*

Proof. Note first that processes have only trivial knowledge about the number of correct or faulty processes in a run, which means that, as far as they know, the number of faulty processes f in any run can go from 0 to $n - 1$. Hence, any protocol P that implements an eventual leader must do so even if $f = n - 1$.

Now, let us consider an infinite run R of P with $f \in [0, n - 1]$. Let us assume, by way of contradiction, that in R some correct process p stops receiving messages from the rest of processes at and after some time τ, but it never becomes its own leader at any time $\tau' \geqslant \tau$. Consider another run R' of P with $f = n - 1$. R' behaves exactly like R up to time τ, and all processes (that were still alive) except p crash at time τ. From the point of view of p these two runs are undistinguishable, and hence p behaves in R' exactly as in R. This implies that it never becomes its own leader at any time $\tau' \geqslant \tau$. This contradicts the requirement that P must implement an eventual leader in R'. □

Let us notice that the above lemma holds even if the process p receives messages *after* time τ'.

Theorem 2. *Let us consider a system of $n \geqslant 3$ processes in which processes have no other initial knowledge on the number of correct or faulty processes than $t = n - 1$, they may know the system membership Π, and there is a pair of directed asynchronous reliable links connecting each pair of distinct processes. Any protocol that implements an eventual leader must have at least $n - f - 1$ eventually timely links in some executions.*

Proof. If $f = n - 1$, the claims follow trivially. For the case $f < n - 1$, the proof is by contradiction as follows. We first assume that there is a protocol P implementing Ω in an asynchronous system with reliable links where only $(n - f - 2)$ links eventually behave timely and f processes fail. We then use the eventual leadership property of Ω to construct an infinite execution of the protocol P with f failures in which this property is not satisfied. Then, protocol P cannot exist.

For the sake of contradiction, assume there is a protocol P implementing Ω in runs with f failures of an asynchronous system with reliable links where only $(n - f - 2)$ links eventually behave timely. This means that there are two correct processes (maybe more), namely p_i and p_j, whose input links are all asynchronous. P thus provides each process p_x, $1 \leqslant x \leqslant n$, with a value LEADER$_x$ (holding the process that p_x currently considers to be the leader). We use LEADER$_x(\tau)$ to denote this value at time τ. We will construct an execution E of P with f failures such that the following both hold:

1) After time $\tau_0 = 0$, E is fault-free (no process fails), i.e., all f faulty processes crash at time $\tau_0 = 0$.

2) There is an infinite sequence of times $\tau_0 < \tau_1 < \tau_2 < \cdots$ such that, for all $k \geqslant 0$, in each interval $(\tau_k, \tau_{k+1}]$ there are two time instants $\tau, \tau' \in (\tau_k, \tau_{k+1}]$ at which at least one of the processes p_i and p_j have different leaders, i.e., $\text{LEADER}_i(\tau) \neq \text{LEADER}_i(\tau')$ or $\text{LEADER}_j(\tau) \neq \text{LEADER}_j(\tau')$.

Clearly, in the execution E of P, two processes disagree on the leader infinitely often and, consequently, the eventual leadership is not satisfied.

As previously described, we define $\tau_0 = 0$, and make f processes fail at this time. We construct the execution E inductively. For $k \geqslant 1$, assume that E is already constructed up to time τ_{k-1} (τ_0 in the base case); we show how to define τ_k and construct the interval $(\tau_{k-1}, \tau_k]$ of the execution E such that item 2) above is satisfied for the value k.

Then, after τ_{k-1} all links behave timely in execution E until some time $\tau > \tau_{k-1}$, at which P has all processes agree on a leader p_{ℓ_k}. This time τ exists by eventual leadership. In particular, it holds that $\text{LEADER}_i(\tau) = \text{LEADER}_j(\tau) = \ell_k$.

Let s_k be a process id such that $s_k \in \{i, j\}$ and $s_k \neq \ell_k$. Such an id exists because $i \neq j$. Then, in execution E, after time τ all links continue behaving timely except the incoming links to p_{s_k}. These links, which are all asynchronous, delay the delivery of all messages until some time $\tau' \geqslant \tau$ at which P makes p_{s_k} be its own leader (i.e., $\text{LEADER}_{s_k}(\tau') = s_k$). This time τ' exists by Lemma 6.

Let us define $\tau_k = \tau'$. For completeness, we force in E that all messages sent in the interval $[\tau_{k-1}, \tau_k)$ and still undelivered at time τ_k to be delivered at that time. By construction it follows that there are two time instants $\tau, \tau' \in (\tau_{k-1}, \tau_k]$ that satisfy $\text{LEADER}_i(\tau) = \text{LEADER}_j(\tau) = \ell_k$ and $\text{LEADER}_{s_k}(\tau') \neq \ell_k$, for $s_k \in \{i, j\}$. Hence, item 2) above is satisfied for the value k.

Repeating this process for every $k > 0$, we construct an infinite sequence of intervals that constitutes the execution E. Hence we obtain an execution E that satisfies items 1) and 2) mentioned above, which completes the proof. □

5 A Communication-Efficient Protocol

This section presents an eventual leader protocol (FJR2) where, after some finite time, a single process sends messages forever. Moreover, no message carries values that increase indefinitely: the counters carried by a message take a finite number of values. This means that, be the execution finite or infinite, both the local

```
Let: $leader()$ be the function returning $\ell$ such that $(-,\ell)$ =$lex\_min(\{(susp\_level_i[j], j)\}_{j \in contenders_i})$
Init: allocate $susp\_level_i[i]$; $susp\_level_i[i] \leftarrow 0$; $hbc_i \leftarrow 0$; $contenders_i \leftarrow \{i\}$; $members_i \leftarrow \{i\}$; $\text{LEADER}_i \leftarrow i$
Task $T1$:
    repeat forever
        $\text{LEADER}_i \leftarrow leader()$;
        $next\_period_i \leftarrow false$;
        while $\text{LEADER}_i = i$ do every $\eta$ time units
            if $(\neg next\_period_i)$ then $next\_period_i \leftarrow true$; $hbc_i \leftarrow hbc_i + 1$ end if;
            broadcast $(heartbeat, i, susp\_level_i[i], \perp, hbc_i)$;
            $\text{LEADER}_i \leftarrow leader()$;
        end while;
        if $(next\_period_i)$ then broadcast $(stop\_leader, i, susp\_level_i[i], \perp, hbc_i)$ end if
    end repeat
Task $T2$:
when $timer_i[j]$ expires:
    $timeout_i[j] \leftarrow timeout_i[j] + 1$; broadcast $(suspicion, i, susp\_level_i[i], j, 0)$;
    $contenders_i \leftarrow contenders_i \setminus \{j\}$
when $(tag\_k, k, sl\_k, silent\_k, hbc\_k)$ is received with $k \neq i$ :
    if $(k \notin members_i)$ then $members_i \leftarrow members_i \cup \{k\}$;
                                   allocate $susp\_level_i[k]$ and $last\_stop\_leader_i[k]$;
                                   $susp\_level_i[k] \leftarrow 0$; $last\_stop\_leader_i[k] \leftarrow 0$;
                                   allocate $timeout_i[k]$ and $timer_i[k]$; $timeout_i[k] \leftarrow \eta$ end if;
    $susp\_level_i[k] \leftarrow \max(susp\_level_i[k], sl\_k)$;
    if $((tag\_k = heartbeat) \wedge last\_stop\_leader_i[k] < hbc\_k))$
            then set $timer_i[k]$ to $timeout_i[k]$; $contenders_i \leftarrow contenders_i \cup \{k\}$ end if;
    if $((tag\_k = stop\_leader) \wedge last\_stop\_leader_i[k] < hbc\_k))$
            then $last\_stop\_leader_i[k] \leftarrow hbc\_k$;
                 stop $timer_i[k]$; $contenders_i \leftarrow contenders_i \setminus \{k\}$ end if;
    if $((tag\_k = suspicion) \wedge (silent\_k = i))$ then $susp\_level_i[i] \leftarrow susp\_level_i[i] + 1$ end if
```

Fig.2. The communication-efficient eventual leader protocol FJR2 (code for p_i).

memory of each process and the message size are finite. The process initial knowledge is limited to (K1), while the network behavior is assumed to satisfy (C1′) and (C2′).

5.1 Description of the Protocol

The protocol FJR2 is described in Fig.2. As FJR1, this protocol is made up of two tasks, but presents important differences with respect to the previous protocol. To guarantee correctness, five sets of statements of the algorithm are defined as being executed in mutual exclusion: Lines 02∼03, Lines 05∼07, and Line 09 in task $T1$, and Lines 11∼12 and Lines 13∼23 in task $T2$.

Local Variables. A first difference is the Task $T1$, where a process p_i sends messages only when it considers it is a leader (Line 04). Moreover, if, after being a leader, p_i considers it is no longer a leader, it broadcasts a message to indicate that it considers locally it is no longer leader (Line 09). A message sent with a tag field equal to *heartbeat* (Line 06) is called a heartbeat message; similarly, a message sent with a tag field equal to *stop_leader* (Line 09) is called a stop_leader message.

A second difference lies in the additional local variables that each process has to manage. Each process p_i maintains a set, denoted $contenders_i$, plus local counters, denoted hbc_i and $last\_stop\_leader_i[k]$ (for each process p_k that p_i is aware of). More specifically, we have:

- The set $contenders_i$ contains the ids of the processes that compete to become the final common leader, from p_i's point of view. So, we always have $contenders_i \subseteq members_i$. Moreover, we also always have $i \in contenders_i$. This ensures that a leader election is not missed since, from its point of view, p_i is always competing to become the leader.
- The local counter hbc_i registers the number of distinct periods during which p_i considered itself the leader. A period starts when $\text{LEADER}_i = i$ becomes true, and finishes when thereafter it becomes false (Lines 04∼13).
- The counter $last\_stop\_leader_i[k]$ contains the greatest hbc_k value ever received in a stop_leader message sent by p_k. This counter is used by p_i to take into account a heartbeat message (Line 18) or a stop_leader message (Line 20) sent by p_k, only if no "more recent" stop_leader message has been received (the notion of "more recent" is with respect to the value of hbc_i associated with and carried by each message).

Messages. Another difference lies in the shape and the content of the messages sent by a process. A message has five fields $(tag\_k, k, sl\_k, silent\_k, hbc\_k)$ whose meaning is the following:

- The field tag_k can take three values: *heartbeat*, *stop_leader* or *suspicion* that defines the type of the message. (Similarly to the previous cases, a message tagged *suspicion* is called a suspicion message. Such a message is sent only at Line 11.)
- The second field contains the id k of the message sender.
- sl_k is the value of $susp\_level_k[k]$ when p_k sent that message. Let us observe that the value of $susp\_level_k[k]$ can be disseminated only by p_k.
- $silent\_k = j$ means that p_k suspects p_j to be faulty. Such a suspicion is due to a timer expiration that occurs at Line 11. (Let us notice that the field $silent\_k$ of a message that is not a suspicion message is always equal to $\perp$.)
- hbc_k: this field contains the value of the period counter hbc_k of the sender p_k when it sent the message. (It is set to 0 in suspicion messages.)

The set of messages tagged *heartbeat* or *stop_leader* defines a single type of message. Differently, there are n types of messages tagged *suspicion*: each pair (*suspicion*, $silent_k$) defines a type.

Process Behavior. When a timer $timer_i[j]$ expires, p_i broadcasts a message indicating it suspects p_j (Line 11)[6], and accordingly suppresses j from $contenders_i$. Together with Line 22, this allows all the crashed processes to eventually disappear from $contenders_i$. When p_i receives a $(tag\_k, k, sl\_k, silent\_k, hbc\_k)$ message, it allocates new local variables if that message is the first it receives from p_k (Lines 13∼16); p_i also updates $susp\_level_i[k]$ (Line 17). Then, the processing of the message depends on its tag.

- The message is a heartbeat message (Lines 18∼19). If it is not an old message (this is checked with the test $last\_stop\_leader_i[k] < hbc\_k$), p_i resets the corresponding timer and adds k to $contenders_i$.
- The message is a stop_leader message (Lines 20∼22). If it is not an old message, p_i updates its local counter $last\_stop\_leader_i[k]$, stops the corresponding timer and suppresses k from $contenders_i$.
- The message is a suspicion message (Lines 23). If the suspicion concerns p_i, it increases accordingly $susp\_level_i[i]$.

The protocol FJR2 is based on the communication-efficient protocol of [19], adapted to the system properties. In particular, FJR2 cannot use point-to-point

[6] The suspicion message sent by p_i concerns only p_j. It is sent by a broadcast primitive only because the model does not offer a point-to-point send primitive. If a point-to-point send primitive was available the broadcast at Line 11 would be replaced by the statement "send $(suspicion, i, susp\_level_i[i], 0)$ to p_j", and all the suspicion messages would then define a single message type. In that case each tag would define a message type. This shows an interesting tradeoff relating communication primitives (one-to-one vs. one-to-many) and the number of message types.

communication and uses broadcast instead. It also has to deal with the fact that the membership is unknown. Finally, a new mechanism to guarantee communication-efficiency and finite memory is introduced by FJR2 (based on stop_leader messages).

5.2 Proof of the Protocol

This subsection proves that 1) the protocol FJR2 described in Fig.2 eventually elects a common correct leader, and 2) no message carries values that indefinitely grow. The proofs assume only (K1) as far the process initial knowledge is concerned. It assumes (C1′) and (C2′) as far as the network behavioral assumptions are concerned.

Lemma 7. *Let p_k be a faulty process. There is a finite time after which the predicate $k \notin contenders_i$ remains permanently true at each correct process p_i.*

Proof. Let p_k and p_i be a faulty process and a correct process, respectively. The only line where a process is added to $contenders_i$ is Line 19. If follows that, if p_i never receives a heartbeat message from p_k, k is never added to $contenders_i$ and the lemma follows for p_k.

So, considering the case where p_i receives at least one heartbeat message from p_k, let us examine the last heartbeat or stop_leader message m from p_k received and processed by p_i. "Processed" means that the message m carried a field hbc_k such that the predicate $last\_stop\_leader_i[k] < hbc\_k$ was true when the message was received. Let us notice that there is necessarily such a message, because at least the first heartbeat or stop_leader message from p_k received by p_i satisfies the predicate.

Due to the very definition of m, there is no other message from p_k such that p_i executes Line 19 or Line 22 after having processed m. There are two cases, according to the tag of m.

- If m is a stop_leader message, p_i executes Line 22 and consequently suppresses definitely k from $contenders_i$.
- If m is a heartbeat message, p_i executes Line 19. This means that it resets $timer_i[k]$ and adds k to $contenders_i$. Then, as no more heartbeat messages from p_k are processed by p_i, $timer_i[k]$ eventually expires and consequently p_i withdraws k from $contenders_i$ (Line 12), and never adds it again (as m is the last processed heartbeat message), which proves the lemma. □

Given a run, let B be the set of correct processes p_i such that the largest value ever taken by $susp\_level_i[i]$ is bounded. Moreover, let M_i denote that value. Let H be the set of correct processes whose all output links with respect to each other correct process are eventually timely. Due to the assumption (C2′), we have $H \neq \varnothing$.

Lemma 8. *$B \neq \varnothing$.*

Proof. The proof consists in showing that $H \subseteq B$. Then, as $H \neq \varnothing$, the lemma follows.

Let p_i be a process in H. $susp\_level_i[i]$ is increased each time p_i receives a suspicion message with $silent\_k = i$ (Line 23). Such a suspicion message can be sent by a process p_j only at Line 11 when $timer_j[i]$ expires. If p_j is faulty it sends a finite number of suspicion messages concerning p_i, and consequently these suspicion messages entail a finite increase of $susp\_level_i[i]$. So, in the following we consider only the case of a process p_j that is correct. The only line where $timer_j[i]$ is set is Line 19, where p_j receives and processes a heartbeat message from p_i. The proof is a case analysis.

- p_i sends a finite number of heartbeat messages. In that case, any correct process p_j receives a finite number $nb[i, j]$ of heartbeat messages from p_i. As (see the previous discussion) the number of suspicion messages that p_j sends to p_i is $\leqslant nb[i, j]$, and the link from p_j to p_i is fair lossy (assumption C1′) and does not duplicate messages, p_i receives a finite number of suspicion messages from each correct p_j. It follows that p_i increases $susp\_level_i[i]$ a finite number of times. (M_i is this number.)
- p_i sends an infinite number of heartbeat messages. We consider here two subcases:
  - There is a time τ after which p_i continuously executes the while loop (Lines 04∼06) in Task $T1$. This means that, after τ, p_i sends forever heartbeat messages with the same hbc_i value every η time units (after τ, it never executes Line 09).

    Let τ', $\tau' \geqslant \tau$, be a time after which the faulty processes have crashed, the links from p_i to the correct processes are timely, and all the stop_leader messages sent by p_i have been received or are lost.

    Let us first observe that any correct process p_j ($\neq p_i$) allocates $timer_j[i]$. Moreover, after τ', p_i sends an infinite number of heartbeat messages carrying the same value hbc_i that is greater than $last\_stop\_leader_j[i]$. As no stop_leader message carrying a value $\geqslant hbc_i$ is ever sent, it follows that p_j processes all these heartbeat messages, i.e., it executes Line 19 and resets $timer_j[i]$ each time it receives such a heartbeat message from p_i.

    It is possible that, after τ', $timer_j[i]$ expires because a heartbeat message has not yet been received by p_j. Each time this occurs, $timeout_j[i]$ is increased, and a suspicion message is sent by p_j to p_i (Line 11). But, as, after τ', the link from p_i to p_j is timely, this can happen only a finite number of times. It follows that any process can send to p_i only a finite number of suspicion messages. There is consequently a time τ'' after which p_i does no longer receive suspicion messages. The value of $susp\_level_i[i]$ at τ'' is then a finite value M_i.

– p_i enters and leaves the while loop but never remains inside forever. This means that p_i sends batches of heartbeat messages. The heartbeat messages sent in the same batch carry the same *hbc_k* value (Line 06), and heartbeat messages of consecutive batches carry increasing *hbc_k* values (Line 05). Moreover, two consecutive batches are separated by the sending of a stop_leader message carrying the same *hbc_k* value as the heartbeat messages of the first of these batches (Line 09). Each batch corresponds to a continuous period during which p_i considers it is the leader. The number of such periods is infinite (otherwise, we would be in the case where p_i sends a finite number of heartbeat messages). We show that (as in the previous item) a process p_j sends a finite number of suspicion messages to p_i.

The timer $timer_j[i]$ of p_j can expire (Line 11) only because, since the last heartbeat message from p_i that entailed the setting of $timer_j[i]$ (at Line 19), p_j has not yet received an appropriate message from p_i: either 1) a heartbeat message (to reset the timer, Line 19), or 2) a stop_leader message (to stop the timer, Line 22) carrying a field *hbc_k* such that $hb\_k > last\_stop\_leader_j[i]$. Each time this occurs, the timeout delay $timeout_j[i]$ is systematically increased (Line 11). Let us also notice that we are in a case where each heartbeat message sent by p_i is followed by another heartbeat or stop_leader message carrying an *hbc_k* value equal to or greater than the previous *hbc_k* values already sent.

Let τ be a time after which the faulty processes have crashed, and the link from p_i to p_j is timely. After τ, the heartbeat or stop_leader messages sent by p_i after each heartbeat message are timely with respect to p_j. It follows that the timer $timer_j[i]$ can expire only a finite number of times, namely, until $timeout_j[i]$ has increased enough to attain the maximal transfer delay experienced by the link from p_i to p_j. This means that p_j sends a finite number of suspicion messages to p_i, which proves the lemma. □

Let $(M_\ell, \ell) = lex\_min(\{(M_i, i) \mid i \in B\})$. The following observation is a direct consequence of the following facts: B does not contain faulty processes (definition), $B \neq \varnothing$ (Lemma 8), and no two processes have the same id (initial assumption).

Observation 1. *There is a single process p_ℓ. Moreover p_ℓ is a correct process.*

Lemma 9. *Let p_i and p_j be two correct processes. There is a finite time after which either* 1) *the predicate $i \notin contenders_j$ is always satisfied or* 2) $(i \in B \Rightarrow susp\_level_j[i] = M_i) \wedge (i \notin B \Rightarrow susp\_level_j[i] \geqslant M_\ell)$.

Proof. Either there is a finite time after which p_j does not receive heartbeat messages from p_i, or p_j receives infinitely many heartbeat messages from p_i. In the former case, either i was never in $contenders_j$ or it is removed by p_j at Line 12 or 22. In the latter case, due to the fact that the link from p_i to p_j is fair lossy (assumption C1$'$), eventually a heartbeat message sent by p_i is received by p_j with $sl\_k = M_i$ if $i \in B$ or $sl\_k \geqslant M_\ell$ if $i \notin B$, and p_j updates accordingly $susp\_level_j[i]$ at Line 17. □

Lemma 10. *There is a time after which p_ℓ executes forever the while loop of its Task $T1$ (Lines 04∼06).*

Proof. For each faulty process p_j, there is a finite time after which the predicate $j \notin contenders_\ell$ remains forever true (Lemma 7). For each correct process p_j, there is a finite time after which $j \notin contenders_\ell$ is always true, or $susp\_level_\ell[j] \geqslant M_\ell$ (this follows from Lemma 9 and the fact that $M_\ell \leqslant M_j$, $\forall\ j \in B$). As $\ell \in contenders_\ell$ is always true, it follows that there is a finite time after which p_ℓ obtains always *true* when it evaluates the predicate $leader()=\ell$, from which we conclude that there is a time after which p_ℓ executes forever the while loop of the Task $T1$ (without ever exiting from this loop). □

Theorem 3. *The protocol described in Fig.2 ensures that, after some finite time, all the correct processes have forever the same correct process p_ℓ as common leader.*

Proof. Due to Observation 1, there is a single process p_ℓ, and that process is correct. Due to Lemma 9, there is a time after which, for each pair of correct processes p_i and p_j we have forever $j \notin contenders_i$ or $susp\_level_i[j] \geqslant M_\ell$. Consequently, to prove the theorem, we have to show that there is a time after which the predicate $\ell \in contenders_i$ remains permanently true at each correct process p_i.

Once p_ℓ has entered the while loop of its Task $T1$ and never exits from it thereafter (due to Lemma 10, this happens), it sends infinitely many heartbeat messages (it sends such an heartbeat message each time it executes Line 06), and from some time these heartbeat messages are such that their *sl_k* field is always equal to M_ℓ. We claim that only a finite number of these heartbeat messages can entail the sending of a suspicion message from p_i to p_ℓ. After this finite number of heartbeat messages have entailed the sending of suspicion messages (Line 11) and the associated suppression of ℓ from $contenders_i$ (Line 12), all the heartbeat messages that are sent subsequently are such that there is no timer expiration and ℓ is added to $contenders_i$ each time such a heartbeat message is received (Line 19). It follows that after some time ℓ belongs permanently to *contenders*$_i$, which proves the theorem.

Proof of the Claim. Let us assume by contradiction that ℓ is suppressed infinitely often from *contenders*$_i$. Each time it is suppressed (Line 12), a suspicion message is sent to p_ℓ (Line 11). This means that an infinite number of suspicion messages are sent by p_i to p_ℓ. As the link from p_i to p_ℓ is fair lossy (C1$'$), p_ℓ receives at least one of these suspicion messages, and increases consequently $susp\_level_\ell[\ell]$ from M_ℓ to $M_\ell + 1$, contradicting the fact that M_ℓ is an upper bound for the values of $susp\_level_\ell[\ell]$. *End of the proof of the claim.*[7] □

5.3 Protocol Optimality

Theorem 4. *There is a time after which exactly one process sends messages forever.*

Proof. The proof is an immediate consequence of the fact that there is a time after which a single correct leader is elected (Theorem 3), the observation that a process sends heartbeat messages only if it considers it is the leader, and the fact that, a finite time after the common leader has been elected, no process sends suspicion messages. □

Theorem 5. *In an infinite run, both the local memory of each process and the size of each message remain finite in the run.*

Proof. Due to Theorem 3, there is a time τ after which a common correct leader is elected. Moreover, due to Theorem 4 there is a time after which only the leader p_ℓ sends messages forever. As then $susp\_level_\ell[\ell]$ remains equal to M_ℓ, and hbc_ℓ keeps on the same value, it follows that both the local memory of each process and the size of each message remain finite, whatever the number of messages that are sent. □

6 Conclusion

This paper has investigated the eventual leader election problem in message-passing systems with weak assumptions on process initial knowledge, communication reliability and synchrony. Two protocols and a lower bound have been presented. The first protocol assumes that each process knows only its id, and a lower bound α on the number of processes that do not crash (it knows neither the number n of processes, nor an upper bound t on the number of faulty processes). This protocol requires the following behavioral properties from the underlying network: the graph made up of the correct processes and fair lossy links is strongly connected, and there is a correct process connected to $(n - f) - \alpha$ other correct processes (where f is the actual number of crashes in the considered run) through eventually timely paths (paths made up of correct processes and eventually timely links). The second protocol is communication-efficient in the sense that, after some time, only the final common leader has to send messages forever. This protocol does not have the knowledge of α, but requires stronger properties from the underlying network: each pair of correct processes is connected by fair lossy links, and there is a correct process whose $n-f-1$ output links to the other correct processes are eventually timely. The lower bound result shows that this number of eventually links is necessary in some executions even if each process initially knows the whole set of identities. Interestingly, the second protocol has another noteworthy property, namely, each value carried by a message is from a finite domain.

As open problems we propose the study of other protocols to implement eventual leader election under other properties of the network. Additionally, we would like to explore how to obtain a stronger lower bound that could relax the $t = n - 1$ constraint of the current bound.

Acknowledgments We would like to thank the referee for his/her constructive comments that help us improve both the content and the presentation of the paper.

References

[1] Chandra T D, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996, 43(2): 225-267.

[2] Raynal M. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News, Distributed Computing Column*, 2005, 36(1): 53-70.

[3] Chandra T D, Hadzilacos V, Toueg S. The weakest failure detector for solving consensus. *Journal of the ACM*, 1996, 43(4): 685-722.

[4] Chen W, Toueg S, Aguilera M K. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 2002, 51(5): 561-580.

[5] Défago X, Urbán P, Hayashibara N, Katayama T. Definition and specification of accrual failure detectors. In *Proc. Int. Conference on Dependable Systems and Networks* (*DSN 2005*), Yokohama, Japan, June 28-July 1, 2005, pp.206-215.

[6] Fetzer C, Raynal M, Tronel F. An adaptive failure detection protocol. In *Proc. the 8th IEEE Pacific Rim Int. Symposium on Dependable Computing* (*PRDC 2001*), Seoul, Korea, Dec. 17-19, 2001, pp.146-153.

[7] Gupta I, Chandra T D, Goldszmidt G S. On scalable and efficient distributed failure detectors. In *Proc. the 20th ACM Symposium on Principles of Distributed Computing* (*PODC 2001*), New Port, USA, Aug. 26-29, 2001, pp.170-179.

[8] Larrea M, Fernández A, Arévalo S. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 2004, 53(7): 815-828.

[9] Wiesmann M, Urbán P, Défago X. An SNMP based failure

[7] One can also conclude that, if any, all the suspicion messages sent by p_i to p_ℓ after p_i has received a heartbeat message from p_ℓ carrying the value M_ℓ, are lost.

detection service. In *Proc. the 25th Int. Symposium on Reliable Distributed Systems* (*SRDS 2006*), IEEE Computer Press, 2006, pp.365-374.

[10] Lamport L. The part-time parliament. *ACM Transactions on Computer Systems*, 1998, 16(2): 133-169.

[11] Schiper A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 1997, 10(9): 149-157.

[12] Mostefaoui A, Raynal M. Leader-based consensus. *Parallel Processing Letters*, 2001, 11(1): 95-107.

[13] Guerraoui R, Raynal M. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 2004, 53(4): 453-466.

[14] Mostefaoui A, Raynal M. Low-cost consensus-based atomic broadcast. In *Proc. the 7th IEEE Pacific Rim Int. Symposium on Dependable Computing* (*PRDC 2000*), Los Angeles, USA, Dec. 18-20, 2000, pp.45-52.

[15] Pedone F, Schiper A. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 2002, 15(2): 97-107.

[16] Fischer M J, Lynch N, Paterson M S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985, 32(2): 374-382.

[17] Mostefaoui A, Raynal M, Travers C. Crash-resilient time-free eventual leadership. In *Proc. the 23rd Int. IEEE Symposium on Reliable Distributed Systems* (*SRDS 2004*), Florianpolis, Brazil, Oct. 18-20, 2004, pp.208-217.

[18] Larrea M, Fernández A, Arévalo S. Optimal implementation of the weakest failure detector for solving consensus. In *Proc. the 19th IEEE Int. Symposium on Reliable Distributed Systems* (*SRDS 2000*), Nürnberg, Germany, Oct. 16-18, 2000, pp.52-60.

[19] Aguilera M K, Delporte-Gallet C, Fauconnier H, Toueg S. On implementing omega with weak reliability and synchrony assumptions. In *Proc. the 22nd ACM Symposium on Principles of Distributed Computing* (*PODC 2003*), Boston, USA, Jul. 13-16, 2003, pp.306-314.

[20] Dwork C, Lynch N, Stockmeyer L. Consensus in presence of partial synchrony. *Journal of the ACM*, 198, 35(2): 288-3238.

[21] Aguilera M K, Delporte-Gallet C, Fauconnier H, Toueg S. Communication efficient leader election and consensus with limited link synchrony. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing* (*PODC 2004*), St. John's Newfoundland, Canada, Jul. 25-28, 2004, pp.328-337.

[22] Malkhi D, Oprea F, Zhou L. Ω meets paxos: Leader election and stability without eventual timeley links. In *Proc. the 19th Int. Symposium on Distributed Computing* (*DISC 2005*), Cracow, Poland, Sept. 26-29, 2005, pp.199-213.

[23] Hutle M, Malkhi D, Schmid U, Zhou L. Chasing the weakest system model for implementing Ω and consensus. Research Report 74/2005, Technische Universität Wien, Institut für Technische Informatik, July 2006.

[24] Mostefaoui A, Mourgaya E, Raynal M, Travers C. A time-free assumption to implement eventual leadership. *Parallel Processing Letters*, 2006, 16(2): 189-208.

[25] Mostefaoui A, Raynal M, Travers C. Time-free and timer-based assumptions can be combined to get eventual leadership. *IEEE Transactions on Parallel and Distributed Systems*, 2006, 17(7): 656-666.

[26] Powell D. Failure mode assumptions and assumption coverage. In *Proc. 22nd Int. Symposium on Fault-Tolerant Computing*, 1992, Boston, USA, pp.386-395.

[27] Jiménez E, Arévalo S, Fernández A. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 2006, 100(2): 60-63.

[28] Attiya H, Bar-Noy A, Dolev D, Peleg D, Reischuk R. Renaming in an asynchronous environment. *Journal of the ACM*, 1990, 37(3): 524-548.

[29] Borowsky E, Gafni E. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symposium on Principles of Distributed Computing* (*PODC 1993*), Ithaca, USA, Aug. 15-18, 1993, pp.41-51.

Antonio Fernández Anta is a senior researcher at the Institute IMDEA Networks in Leganés, Spain, on leave from his position of professor at the Universidad Rey Juan Carlos. Previously, he was on the faculty of the Universidad Politécnica de Madrid. He graduated in computer science from the Universidad Politécnica de Madrid in 1991. He got a Ph.D. degree in computer science from the University of Southwestern Louisiana in 1994 and was a postdoc at the Massachusetts Institute of Technology from 1995 to 1997. His is a senior member of the IEEE and the ACM. His research interests include data communications, computer networks, distributed processing, algorithms, and discrete and applied mathematics.

Ernesto Jiménez graduated in computer science from the Universidad Politécnica de Madrid, Spain, and got a Ph.D. degree in computer science from the University Rey Juan Carlos, Spain, in 2004. He is currently an associate professor at the Universidad Politécnica de Madrid.

Michel Raynal is a professor of computer science at the University of Rennes, France. His main research interests are the basic principles of distributed computing systems. He is a world leading researcher in the domain of distributed computing. He is the author of numerous papers on distributed computing and is well-known for his distributed algorithms and his books on distributed computing. He has chaired the program committee of the major conferences on the topic, such as the ICDCS, the DISC, the SIROCCO, and OPODIS. He has also served on the program committees of many international conferences, and is the recipient of several "Best Paper" awards (ICDCS 1999, 2000 and 2001, SSS 2009, Europar 2010). He has been invited by many universities all over the world to give lectures on distributed computing. His h-index is 45. He has recently written two books published by Morgan & Claypool: "Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems" (June 2010) and "Fault-Tolerant Agreement in Synchronous Distributed Systems" (September 2010).