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

Antonio Fernández, Ernesto Jiménez, Michel Raynal

▶ To cite this version:

Antonio Fernández, Ernesto Jiménez, Michel Raynal. Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony. [Research Report] PI 1770, 2005, pp.19. inria-00000913

HAL Id: inria-00000913
https://inria.hal.science/inria-00000913v1

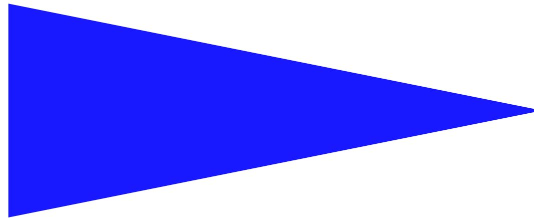
Submitted on 8 Dec 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE
N° 1770



EVENTUAL LEADER ELECTION WITH WEAK ASSUMPTIONS ON INITIAL KNOWLEDGE, COMMUNICATION RELIABILITY, AND SYNCHRONY

ANTONIO FERNÁNDEZ   ERNESTO JIMÉNEZ   MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

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

Antonio Fernández* Ernesto Jiménez** Michel Raynal***

Systèmes communicants

Publication interne n ° 1770 — Décembre 2005 — 19 pages

Abstract: This report 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 the 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 $t - f$ 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 output links to the rest of correct processes have to be eventually timely. The lower bound result shows that this a necessary requirement. This protocol enjoys also the property that each messsage is made up of several fields, each of which taking values from a finite domain.

Key-words: Asynchronous message-passing system, Eventually timely link, Fair lossy link, Eventual leader election, Failure detector, Message loss, Network behavior, Omega leader oracle, Process crash, Process initial knowledge.

(Résumé : tsvp)

* LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain, anto@gsyc.escet.urjc.es
** EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain ernes@eui.upm.es
*** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Election d'un leader avec des hypothèses faibles concernant la connaissance initiale, le synchronisme et la fiabilité

Résumé : Ce rapport présente deux protocoles qui élisent un leader inéluctable dans un système réparti défini par des hypothèses faibles sur connaissance initiale des processus, le synchronisme et la fiabilité des communcations. Un théorème d'impossibilité est également présenté.

Mots clés : Systèmes répartis asynchrones, Tolérance aux fautes, Crash de processus, Oracle oméga, canal équitable, Perte de message, Détection de fautes, Leader inéluctable.

1 Introduction

Leader oracle: motivation Basic services at the core of a lot of fault-tolerant protocols encountered in asynchronous distributed systems are failure detectors [4, 22]. Among them, the class of *leader* failure detectors is one of the most important. This class, usually denoted Ω, 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. An Ω oracle provides the processes with a *leader primitive* that outputs a process id each time it is called, and satisfies the following eventual leadership property: eventually all its invocations return the same id, that id being the identity of a correct process (a process that does not commit failures). Such an oracle is very weak. This means that a correct leader is eventually elected, but there is no knowledge on when this common leader is elected; moreover, several leaders (that can be correct processes or not) can possibly co-exist before this occurs.

The oracle class Ω has several noteworthy features. A fundamental one lies in the fact that, despite its very weak definition, it is powerful enough to allow solving fundamental problems such as the consensus problem [5]. Moreover, it has even been shown that it is the weakest class of failure detectors that allows solving that problem (assuming a majority of correct processes) [5][1]. The liveness property of the well-known Paxos algorithm is based on such a leader oracle [11][2]. Other leader-based consensus protocols can be found in [9, 18].

Another major feature of Ω lies in the fact that it allows designing *indulgent* protocols [8]. Let P be an oracle-based protocol that produces outputs, and PS be the safety property satisfied by its outputs. P is *indulgent with respect to its underlying oracle* if, whatever the behavior of the oracle, its outputs never violate the safety property PS. This means that each time P produces outputs, those are correct. Moreover, P always produces outputs when the underlying oracle meets its specification. The only case where P can be prevented from producing outputs is when the underlying oracle does not meet its specification. (Let us notice that it is still possible that P produces outputs despite the fact that its underlying oracle does not work correctly.) Interestingly, Ω is a class of oracles that allows designing indulgent protocols [8, 9].

Unfortunately, Ω cannot be implemented in pure asynchronous distributed systems where processes can crash. (Such an implementation would contradict the impossibility to solve consensus in such systems [7]. A direct proof of the impossibility to implement Ω in pure crash-prone asynchronous systems can be found in [19].) But thanks to indulgence, this is not totally bad news. More precisely, as Ω makes possible the design of indulgent protocols, it is interesting to design "approximate" protocols that do their best to implement Ω on top of the asynchronous system itself. The periods during which their best effort succeeds in producing a correct implementation of the oracle (i.e., there is a single leader and it is alive) are called "good" periods (and then, the upper layer Ω-based protocol produces outputs and those are correct). During the other periods (sometimes called "bad" periods, e.g., there are several leaders or the leader is a crashed process), the upper layer Ω-based protocol never produces erroneous outputs. The only bad thing that can then happen is that this protocol can be prevented from producing outputs, but when a new long enough good period appears, the upper layer Ω-based protocol can benefit from that period to produce an output.

A main challenge of asynchronous fault-tolerant distributed computing is consequently to identify properties that are at the same time "weak enough" in order to be satisfied "nearly always" by the underlying asynchronous system, while being "strong enough" to allow implementing Ω during the "long periods" where they are satisfied.

Related work The very first implementations of Ω in crash-prone asynchronous distributed systems considered a fully connected communication network where all links are bidirectional, reliable and eventually timely (i.e., there is a time τ_0 after which there is a bound δ -possibly unknown- such that, for any time $\tau \geq \tau_0$, a message sent at time τ is received by time $\tau + \delta$) [13].

[1] Let us remind that, while consensus can be solved in synchronous systems despite Byzantine failures of less than one third of the processes [12], it cannot be solved in asynchronous distributed systems prone to even a single process crash [7].

[2] It is important to notice that the the first version of the Paxos algorithm dates back to 1989, i.e., before the Ω formalism was introduced and investigated.

This "eventually timely links" approach has been refined to obtain weaker constraints. It has been shown in [1] 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 implementing Ω is not communication-efficient in the sense that it requires that all the correct processes send messages forever. It is also shown in [1] that, if additionally there is a correct process whose input and output links are fair lossy, 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 the notion of *communication-efficiency* introduced in [1] is an optimality notion, as, 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 "eventually timely t-source" has been introduced in [2]. Such a source is a correct process that has t eventually timely output links (where t is the maximal number of process crashes). It is shown that such a weak assumption is strong enough for implementing Ω. If the other links are fair lossy, the proposed protocol requires the correct processes to send messages forever. A second protocol is presented that is communication-efficient when additionally the links are reliable and t links are timely (only t links have then to carry messages forever).

Another direction has been recently investigated in [14] where the notion of "eventual t-accessibility" is introduced. A process p is t-accessible at some time τ if there is a set Q of t processes q such that a message broadcast by p at τ receives a response from all the processes of Q by time $\tau + \delta$ (where δ is a bounded value known by the processes). This notion requires a majority of correct processes. Its interest lies in 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. A protocol building Ω when there is a process that is eventually t-accessible forever, and all other links are fair-lossy is described in [14].

A protocol based on a totally different approach to build Ω is described in [19]. It uses the time-free assumption proposed and investigated in [15]. That approach does not rely on timing assumptions and timeouts. It uses explicitly the values of n (the total number of processes) and t (the maximal number of processes that can crash), and consists in stating a property on the message exchange pattern that, when satisfied, allows implementing Ω.

Assuming that each process can broadcast queries and then, for each query, wait for the corresponding responses, let us say that a response to a query is a *winning* if it arrives among the first $(n-t)$ responses to that query (the other responses to that query are called *losing* responses; they can be slow, lost or never sent because their sender has crashed). It is shown in [19] that Ω can be built as soon as the following behavioral property is satisfied: "There are a correct process p and a set Q of $(t+1)$ processes such that eventually the response of p to each query issued by any $q \in Q$ is always a winning response (until -possibly- the crash of q)." When $t = 1$, this property becomes: "There is a link connecting two processes that is never the slowest (in terms of transfer delay) among all the links connecting these two processes to the rest of the system." A probabilistic analysis for the case $t = 1$ shows that such a behavioral property on the message exchange pattern is practically always satisfied [15]. This approach has been extended to dynamic systems in [20] (systems where processes can dynamically enter or leave the system).

Another approach to build Ω is the "reduction" approach. This is a theoretical approach whose aim is to build Ω from other failure detector classes. Let us consider the failure detector class $\diamond\mathcal{S}_x$ introduced in [3, 23]. This class includes all the failure detectors that provide each process p with a set $suspected_p$ containing process ids and satisfying the following properties. *Completeness:* eventually the set $suspected_p$ of each correct process p permanently includes the ids of all the crashed processes; *Limited scope eventual weak accuracy:* there is a time after which, there is a correct process that never appears in the sets of x (correct or faulty) processes. $\diamond\mathcal{S}_n$ corresponds to the class $\diamond\mathcal{S}$ introduced in [4]. It is shown in [3] that, assuming reliable communication, it is possible to build $\diamond\mathcal{S}$ from $\diamond\mathcal{S}_x$ if and only if $x > t$. Moreover, there are protocols that build Ω in crash-prone asynchronous distributed systems equipped with a failure detector of the class $\diamond\mathcal{S}$ [5, 6, 16]. A stacking of the previous protocols provides a protocol building Ω in an asynchronous system equipped with $\diamond\mathcal{S}_{t+1}$ (this is a "reduction" of Ω to $\diamond\mathcal{S}_{t+1}$).

As indicated, this approach is mainly theoretic: its aims is to investigate, compare and rank the computability power of failures detector classes. One of its most important results is the fact that the classes Ω and $\diamond\mathcal{S}$ are equivalent: given a failure detector of any of these classes, it is possible to build a failure detector of the other class [5, 6, 16].

Content of the paper: Weak reliability and synchrony assumptions All the previous protocols implicitly assume that each process initially knows the identity of each other process. It is shown in [10] that this assumption is a necessary requirement for the classes Ω and $\diamond\mathcal{S}$ to be equivalent. Actually, $\diamond\mathcal{S}$ cannot bre built in a system where the initial knowledge of each process is limited to its own identity (if a process crashes before the protocol starts, there is no way for the other processes to learn its id and suspect it). This observation makes Ω more attractive than $\diamond\mathcal{S}$ as its implementation can require weaker assumptions. This paper investigates the implementation of Ω in asynchronous systems that satisfy rather weak assumptions on the initial knowledge of each process, and the behavior of the underlying network. Two protocols and an impossibility result are presented.[3].

The first protocol assumes the following initial knowledge assumptions:

- (K1) A process knows initially neither n, nor t, nor the id of the other processes. It only knows its own id, and the fact that the ids are totally ordered and no two processes have the same id.
- (K2) Each process initially knows the lower bound (denoted α) on the number of correct processes. This means that all but α processes can crash in any run R (α can be seen as the differential value $n-t$).

This protocol is designed 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 $reach(p)$ 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 $|reach(p)| \geq t-f$, where f is the number of actual crashes during the run R.

The design principles of the protocol based on the previous assumptions are the following. As t is an upper bound on the number of process crashes, it is relatively simple to design a leader protocol for the runs in which exactly t processes crash, as, once t processes have crashed, the system cannot experience more crashes (it is then fault-free). The protocol is based on that simple principle: the more processes have crashed, the simpler to elect a leader, and the process that is eventually elected as the final common leader is the process that is the least suspected (this "technique" is used in many leader protocols). Interestingly, this protocol tolerates message duplication.

Then the paper presents an impossibility result that consists in a lower bound theorem. That theorem states that, in any asynchronous system where the initial knowledge of any process includes neither t nor α, there is no leader protocol in the runs where less than $n-1$ links eventually behave in a timely manner.

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

- (C1'): Each pair of processes that are correct in R is connected by (typed) fair lossy channels (one in each direction).

[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.

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

This protocol guarantees that after some 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 memories and the size of the messages remain finite. Differently from the first protocol, this protocol assumes that no link duplicates messages. Its design combines new ideas with ideas used in [1, 2, 10].

To our knowledge, [10] is the only paper that has proposed a leader election protocol for processes that only know their own identity (K1). The first leader election protocol presented in this paper is the first that combines this weak assumption with knowledge of α, allowing weaker network behavioral properties. The second protocol is the first that achieves communication efficiency with assumption (K1).

These protocols show interesting tradeoffs between their requirements ((K2,C1,C2) vs (C1',C2')), and the additional communication-efficient property they provide or not. A problem that remains open consists in designing (or showing the impossibility of designing) a communication-efficient protocol relying on network assumptions weaker than (C1',C2').

Roadmap The paper is made up of 6 sections. Section 2 presents the distributed system model. Section 3 presents the first protocol and proves it is correct. Section 4 states and proves the lower bound result. Section 5 presents the communication-efficient protocol. Finally, Section 6 concludes the paper.

2 Distributed System Model

2.1 Synchronous Processes with Crash Failures

The system is made up of a finite set Π of n processes. Each process p_i has an id. The process ids are totally ordered (e.g., they are integers), but need not be consecutive. Sometimes we also use p or q to denote processes.

As indicated in the introduction, initially, a process p_i knows its own id (i) and the fact that no two processes have the same id. 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 ($1 \leq t < n$); f denotes the number of actual crashes in a given run ($0 \leq f \leq t$). A process knows neither n nor t. The first protocol (only) requires that each process initially knows the lower bound $\alpha = n - t$ on the number of correct processes.

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. To simplify the presentation, and without loss of generality, we assume in the following that local processing takes no time. Only message transfer take time.

2.2 The Communication Network

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

Individual link behavior Each message sent by a process is assumed to be unique. 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 *eventual timely* if there is a time τ_0 and a bound δ such that each message sent by p to q at any time $\tau \geq \tau_0$ is received by q by time $\tau + \delta$ (τ and δ are not a priori known and can never be known).

- 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 eventual 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.

Communication primitive Since processes do not know the id of the other processes, they cannot send point-to-point message to them. Instead, 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.

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, $reach(p)$ (introduced in the first section) is the subset of correct processes q ($q \neq p$) that can be reached from p in the graph G^R_{ET}. (This means that there is a path made up of eventually timely links and correct processes from p to each $q \in reach(p)$.)

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 channels (one in each direction).

- (C2): There is (at least) one correct process p such that $|reach(p)| \geq t - f$.

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

Let us observe that the property (C2) is always satisfied in the runs where $f = t$ (the maximum number of processes allowed to crash effectively crash). Moreover, (C1') and (C2') are stronger than (C1) and (C2), respectively.

2.3 The Class Ω of Oracles

Ω has been defined informally in the introduction. A *leader* oracle is a distributed entity that provides the processes with a function *leader*() that returns a process id each time it is invoked. A unique correct process 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. A leader oracle satisfies the following property:

- Eventual Leadership: There is a time τ and a correct process p such that any invocation of *leader*() issued after τ returns p.

Ω-based consensus algorithms are described in [9, 11, 18] for asynchronous systems where a majority of processes are correct ($t < n/2$). Such consensus algorithms can then be used as a subroutine to solve other problems such as atomic broadcast (e.g., [4, 11, 17, 21]).

3 A Leader Election Protocol

Assuming that each process knows its identity (K1), the lower bound α on the number of correct processes (K2), and that all the processes have distinct and comparable identities, the protocol that follows 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 aim is for a process to elect as its current leader a process that is alive and is perceived 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 Figure 1. It is composed of two tasks. 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.

- $members_i$ 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.
  $silent_i$ 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 invocation of *leader()* in Task $T2$).
  The variable $suspected_by_i[j]$ 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 $\emptyset$ for a new observation period.
- sn_i is a local counter used to generate the increasing 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 and updates accordingly to_reset_i to $\emptyset$.

The second task is devoted to the management of the three events that can locally happen: local call to *leader*(), timer expiration and message reception. The code associated with the two first events 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 $sn' \geq sn_k$. 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 $state_msg$.

```
Init: allocate susp_level_i[i] and suspected_by_i[i]; susp_level_i[i] ← 0; suspected_by_i[i] ← ∅;
        members_i ← {i}; to_reset_i ← ∅; silent_i ← ∅; sn_i ← 0;
        state_i ← {(i, sn_i, {(susp_level_i[i], i)}, silent_i)}      % initial knowledge (K1) %
------------------------------------------------------------------------------------------
Task T1:
    repeat forever every η time units
(01)    sn_i ← sn_i + 1;
(02)    for each j ∈ silent_i do suspected_by_i[j] ← suspected_by_i[j] ∪ {i} end for;
(03)    for each j ∈ members_i such that |suspected_by_i[j]| ≥ α do % initial knowledge (K2) %
(04)                              susp_level_i[j] ← susp_level_i[j] + 1; suspected_by_i[j] ← ∅ end for;
(05)    replace (i, −, −, −) in state_i by (i, sn_i, {(susp_level_i[j], j) | j ∈ members_i}, silent_i);
(06)    broadcast (state_i);
(07)    for each j ∈ to_reset_i do set timer_i[j] to timeout_i[j] end for; to_reset_i ← ∅
    end repeat
------------------------------------------------------------------------------------------
Task T2:
when leader() is invoked by the upper layer:
    return (ℓ such that (−, ℓ) = lex_min({(susp_level_i[j], j)}_{j ∈ members_i}))

when timer_i[j] expires:
(08)    timeout_i[j] ← timeout_i[j] + 1; silent_i ← silent_i ∪ {j}

when state_msg is received:
(09)    let K = { (k, sn_k, cand_k, silent_k) |
              (k, sn_k, cand_k, silent_k) ∈ state_msg  ∧  ∄ (k, sn′, −, −) ∈ state_i with sn′ ≥ sn_k };
(10)    for each (k, sn_k, cand_k, silent_k) ∈ K do
(11)        if k ∈ members_i then replace (k, −, −, −) in state_i by (k, sn_k, cand_k, silent_k);
(12)                              stop timer_i[k]; to_reset_i ← to_reset_i ∪ {k}; silent_i ← silent_i \ {k}
(13)                        else  add (k, sn_k, cand_k, silent_k) to state_i;
(14)                              allocate susp_level_i[k], suspected_by_i[k], timeout_i[k] and timer_i[k];
(15)                              susp_level_i[k] ← 0; suspected_by_i[k] ← ∅; timeout_i[k] ← η;
(16)                              members_i ← members_i ∪ {k}; to_reset_i ← to_reset_i ∪ {k}
            end if
        end for;
(17)    for each (k, sn_k, cand_k, silent_k) ∈ K do
(18)        for each (sl, ℓ) ∈ cand_k do susp_level_i[ℓ] ← max(susp_level_i[ℓ], sl) end for;
(19)        for each ℓ ∈ silent_k do suspected_by_i[ℓ] ← suspected_by_i[ℓ] ∪ {k} end for
        end for
```

Figure 1: An eventual leader protocol (code for p_i)

3.2 Proof of the Protocol

Considering that each processing block (body of the loop in Task $T1$, local call to *leader*(), timer expiration and message reception managed in Task $T2$) is executed atomically, we have $\big(j \in members_i\big)$ iff $\big((j,-,-,-) \in state_i\big)$ iff $\big(suspected_by_i[j]$ and $suspected_by_i[j]$ are allocated$\big)$. We also have $\big(timer_i[j]$ and $timeout_i[j]$ are allocated$\big)$ iff $\big(j \in members_i \setminus \{i\}\big)$. 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 08 of Figure 1-).

The proof considers an arbitrary run R. Let L be the set that contains all the processes p_i that are correct in R and $|reach(i)| \geq t - f$. By property (C2) and by assumption $L \neq \emptyset$.

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 $sn' \geq sn$.*

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 $sn' \geq sn$ belongs to $state_i$.

If $state_i$ does not contain a quadruple $(k, -, -, -)$, p_i adds $(k, sn, -, -)$ to $state_i$ (Line 13). Otherwise, p_i replaces in $state_i$ the old quadruple $(k, -, -, -)$, by the new one $(k, sn, -, -)$ (Line 11). 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 09 and 11). 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_{FL} is strongly connected, it follows that each correct process eventually receives the quadruple $(k, sn', -, -)$ or a quadruple $(k, sn'', -, -)$ with $sn'' > sn'$. $\Box_{Lemma\ 1}$

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

Proof Let us first observe that, in order to include i into $silent_j$, $timer_j[i]$ has to expire (Line 08). So, to prove the lemma, we show that 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 05-06), each containing a quadruple $(i, sn_i, -, -)$. As G^R_{FL} is strongly connected, it follows that there is a time after which each process p_j such that $j \in reach(i)$ (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 $timer_j[i]$ and $timeout_j[i]$ (Line 14).

The consecutive quadruples $(i, sn_i, -, -)$ periodically sent by p_i within $state_i$ messages (Lines 05-06) contain strictly increasing sequence numbers (Line 01). It follows that any p_j such that $j \in reach(i)$ receives (through the paths in G^R_{FL}) an infinite number of messages carrying quadruples $(i, sn_i, -, -)$, and that infinitely often these arrive when $(i, sn', -, -) \in state_j$ with $sn_i > sn'$. It follows that infinitely often p_j executes $silent_j \leftarrow silent_j \setminus \{i\}$ (Line 12).

The rest of the proof consists in showing that there is a time after which $timer_j[i]$ 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 $timeout_j[i]$ at time τ_0, and d the distance from p_i to p_j in G^R_{ET} $(1 \leq d \leq n-1)$.

- Case 1. If $\Delta^{\tau_0}_{i,j} > d(\eta + \delta)$, due to the very definition of G^R_{ET}, $timer_j[i]$ can no longer expire because it is regularly stopped at Line 12 before $\Delta^{\tau_0}_{i,j}$ time units have elapsed since its last resetting at Line 07 (let us notice that different quadruples $(i, -, -, -)$ can take distinct paths in G^R_{ET} from p_i to p_j).
- Case 2. If $\Delta^{\tau_0}_{i,j} \leq d(\eta + \delta)$, it is possible that $timer_j[i]$ expires before a quadruple $(i, sn_i, -, -)$ with $sn_i > sn'$ (where $(i, sn', -, -) \in state_j$) arrives at p_j. That process consequently increases $timeout_j[i]$ (Line 08). 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.

$\Box_{Lemma\ 2}$

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

Proof Let us first observe that any local variable $susp_level_k[\ell]$ can be updated only at Line 04 or Line 18, and can only increase. p_k being any process, let us examine its local variable $susp_level_k[i]$ where i is such that p_i belongs to L (i.e., p_i is a correct process such that $|reach(i)| \geq t - f$). Due to Lemma 2, there is a time after which there is a set of at least $|reach(i)|$ correct processes p_j whose local predicate $i \in silent_j$ 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 a finite time τ after which at most β processes p_ℓ can send $state_\ell$ messages including $(\ell, -, -, silent_\ell)$ with $i \in silent_\ell$. These β processes can be all the processes but the $t - f$ processes of $reach(i)$, p_i itself (as already noticed, i never belongs to $silent_i$), and the f faulty processes, i.e., $\beta \leq n - (t - f) - 1 - f = (n - t) - 1 = \alpha - 1$. It follows that after some finite time (when all the quadruples $(x, -, -, silent_x)$ with $i \in silent_x$ disseminated from the $(t - f) + f$ processes p_x have arrived or are lost), no process p_k can increase its local variable $susp_level_k[i]$ at Line 04. 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 $susp_level_k[i]$ of all the correct processes become equal (Line 18) and keep forever their common value (denoted SL_i), which proves the lemma. $\square_{Lemma\ 3}$

Lemma 4 *Let B be the set of processes p_i such that $susp_level_k[i]$ remains bounded at some correct process p_k. (1) $B \neq \emptyset$. (2) $\forall i \in B$, the local variables $susp_level_k[i]$ of all the correct processes p_k remain forever equal to the same bounded value (denoted SL_i).*

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

For the processes in L, the second item is proved by Lemma 3. Let p_i be a process in $B \setminus L$. The fact that the local variables $susp_level_k[i]$ of all the correct processes p_k remain forever equal is an immediate consequence of Line 18 and the gossiping mechanism used to propagate the quadruples $(i, sn_i, -, -)$ (Lemma 1). $\square_{Lemma\ 4}$

Lemma 5 *Let p_i be a faulty process. Either all the correct processes p_j are such that $i \notin members_j$ forever, or their local variables $susp_level_j[i]$ increase indefinitely.*

Proof If no correct process p_j ever receives a message including a quadruple $(i, sn_i, -, -)$, then the variable $members_j$ of all the correct processes trivially remain such that $i \notin members_j$.

So, let us consider the case where at least one correct process p_k receives a quadruple $(i, sn_i, -, -)$ 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 $(i, sn_i, -, -)$ 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 $(i, sn'_i, -, -)$ with $sn'_i > sn_i$ (this follows from the definition of $(i, sn_i, -, -)$ that is the last quadruple sent by p_i). Each correct process p_k then sets $timer_k[i]$ to $timeout_k[i]$. As no quadruple $(i, sn'_i, -, -)$ with $sn'_i > sn_i$ is ever received, it follows that (1) $timer_k[i]$ expires and p_k adds p_i to $silent_k$ (at Line 08); and (2) i is never withdrawn from $silent_k$ (at Line 12). 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 $\geq \alpha$ (at Line 19) infinitely often. Consequently, each correct process p_j infinitely often increases $susp_level_j[i]$ (Line 04) which proves the lemma. $\square_{Lemma\ 5}$

Theorem 1 *The protocol described in Figure 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 eventually output the same process id each time they invoke *leader*(), and that id is the identity of a correct process. $\square_{Theorem\ 1}$

4 A Lower Bound

The previous protocol requires knowledge of α (K2) which allows weak knowledge on process identifiers (K1) and weak behavioral assumptions from the underlying network ((C1) and (C2)). However, the protocol is not communication-efficient as each correct process has to send messages forever. Several leader protocols that are communication-efficient have been presented in the literature (e.g., [1]) when each process initially knows the whole set of identities. The next section presents a communication-efficient leader election protocol where the initial knowledge of a process is limited to its id (it has no knowledge of α). This protocol requires the

network behavioral assumptions (C1') and (C2') that are stronger than (C1) and (C2). Before describing this protocol, this section shows a lower bound on the network behavior associated with all the protocols where the process initial knowledge does not include t nor α. It is shown here that the number $n - f - 1$ of eventually timely links required (C2') by the protocol is in fact optimal.

Theorem 2 *Let us consider a system of $n \geq 3$ processes where the initial knowledge of each process includes neither t nor $\alpha = n - t$ (it can include the whole set of process ids, though), and there is a pair of directed asynchronous reliable links connecting each pair of distinct processes. Any protocol that implements an eventual leader in executions with f failures requires at least $n - f - 1$ eventually timely links.*

Proof The proof is by contradiction and is 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 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 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, $1 \leq x \leq n$, with a value $leader_x$ (holding the processes 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. E is fault-free (no process fails) after time $\tau_0 = 0$.
2. There is an infinite sequence of times $\tau_1 < \tau_2 < \tau_3 < \cdots$ such that, for all $k > 0$, in each interval $(\tau_k, \tau_{k+1}]$ there are two time instants $\tau, \tau' \in (\tau_k, \tau_{k+1}]$ at which the processes p_i and p_j have different leaders, i.e., $leader_i(\tau) \neq leader_j(\tau')$.

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

For simplicity, we define $\tau_0 = 0$, and make f processes fail at this time. We construct the execution E inductively. For $k \geq 1$, 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.

Consider another execution E_k of P that is identical to E up to time τ_{k-1} and in which no process crashes for some time after time τ_{k-1}. By eventual leadership, there is some time $\tau > \tau_{k-1}$ after which P has all processes agree on a leader p_{l_k}. In particular, $leader_i(\tau) = leader_j(\tau) = l_k$. Let us define the process id s_k as follows: $s_k = i$ if $l_k = j$, and $s_k = j$ otherwise (here, the important point is that $s_k \neq l_k$). Then, in E_k all processes except p_{s_k} crash after τ. By eventual leadership, there is some time $\tau' > \tau$ after which P has p_{s_k} elect itself leader, i.e., $leader_{s_k}(\tau') = s_k$.

Let $\tau_k = \tau'$, and make E behave in the interval $(\tau_{k-1}, \tau_k]$ as follows. In the interval $(\tau_{k-1}, \tau]$ it behaves exactly as in E_k. In the interval $(\tau, \tau_k]$, however:

- No process crashes but all the messages sent in the interval to p_{s_k} by the rest of the processes remain undelivered at the end of the interval (namely, at τ_k). Messages sent to p_{s_k} in $[0, \tau]$ but not delivered in that interval are delivered (or not) in $(\tau, \tau_k]$ of E exactly as they are (or not) in that interval of E_k.
- Process p_{s_k} behaves exactly as in the interval $(\tau, \tau_k]$ of E_k. Moreover, the delivery of all the messages it sends during $(\tau, \tau_k]$ are delayed until after τ_k.

The above imply that no process can distinguish executions E_k and E through time τ_k. It follows that we have $leader_{s_k}(\tau_k) = s_k$ and $\forall x \neq s_k$, $leader_x(\tau) = l_k$. Consequently, there are two time instants $\tau 1$ and $\tau 2$ in the interval $(\tau_{k-1}, \tau_k]$ such that $leader_i(\tau 1) \neq leader_j(\tau 2)$.

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. $\Box_{Theorem\ 2}$

5 A Communication-Efficient Protocol

As announced previously, this section presents an eventual leader protocol 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 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 is described in Figure 2. As the protocol described in Figure 1, this protocol is made up of two tasks, but presents important differences with respect to the previous protocol.

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 01). 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 04). A message sent with a tag field equal to *heartbeat* (Line 03) is called a heartbeat message; similarly, a message sent with a tag field equal to *stop_leader* (Line 04) 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 allows not to miss electing a leader as, 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 $leader() = i$ becomes true, and finishes when thereafter it becomes false (Lines 01-04).
- 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 12) or a stop_leader message (Line 14) 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 05.)
- 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 05. (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 messages. Differently, there are n types of messages tagged *suspicion*: each pair (*suspicion*, $silent_k$) defines a type.

```
Init: allocate susp_level_i[i]; susp_level_i[i] ← 0;
        hbc_i ← 0; contenders_i ← {i}; members_i ← {i}
--------------------------------------------------------------------------
Task T1:
      repeat forever
          next_period_i ← false;
(01)      while leader() = i do every η time units
(02)          if (¬next_period_i) then next_period_i ← true; hbc_i ← hbc_i + 1 endif;
(03)          broadcast (heartbeat, i, susp_level_i[i], ⊥, hbc_i)
          end while;
(04)      if (next_period_i) then broadcast (stop_leader, i, susp_level_i[i], ⊥, hbc_i) end if
      end repeat
--------------------------------------------------------------------------
Task T2:
when leader() is invoked:
      return (ℓ such that (−, ℓ) =lex_min({(susp_level_i[j], j)}_{j∈contenders_i}))

when timer_i[j] expires:
(05)  timeout_i[j] ← timeout_i[j] + 1; broadcast (suspicion, i, susp_level_i[i], j, 0);
(06)  contenders_i ← contenders_i \ {j}

when (tag_k, k, sl_k, silent_k, hbc_k) is received with k ≠ i :
(07)  if (k ∉ members_i) then members_i ← members_i ∪ {k};
(08)                          allocate susp_level_i[k] and last_stop_leader_i[k];
(09)                          susp_level_i[k] ← 0; last_stop_leader_i[k] ← 0;
(10)                          allocate timeout_i[k] and timer_i[k]; timeout_i[k] ← η end if;
(11)  susp_level_i[k] ← max(susp_level_i[k], sl_k);
(12)  if ((tag_k= heartbeat) ∧ last_stop_leader_i[k] < hbc_k))
(13)          then set timer_i[k] to timeout_i[k]; contenders_i ← contenders_i ∪ {k} endif;
(14)  if ((tag_k = stop_leader) ∧ last_stop_leader_i[k] < hbc_k))
(15)          then last_stop_leader_i[k] ← hbc_k;
(16)               stop timer_i[k]; contenders_i ← contenders_i \ {k} endif;
(17)  if ((tag_k = suspicion) ∧ (silent_k = i)) then susp_level_i[i] ← susp_level_i[i] + 1 endif
```

Figure 2: A communication-efficient eventual leader protocol (code for p_i)

Process behavior When a timer $timer_i[j]$ expires, p_i broadcasts a message indicating it suspects p_j (Line 05)[4], and accordingly suppresses j from $contenders_i$. Together with Line 16, 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 07-10); p_i also updates $susp_level_i[k]$ (Line 11). Then, the processing of the message depends on its tag.

- The message is a heartbeat message (Lines 12-13). 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 14-16). 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 17). If the suspicion concerns p_i, it increases accordingly $susp_level_i[i]$.

5.2 Proof of the Protocol

This section proves that (1) the protocol described in Figure 2 eventually elects a common correct leader, and (2) no message carries values that indefinitely grow. The proofs assumes only (K1) as far the process

[4]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 05 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.

initial knowledge is concerned. It assumes (C1') and (C2') as far as the network behavioral assumptions are concerned.

Lemma 6 *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 13. 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 13 or Line 16 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 16 and consequently suppresses definitely k from $contenders_i$.
- If m is a heartbeat message, p_i executes Line 13. 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 06), and never adds it again (as m is the last processed heartbeat message), which proves the lemma.

$\square_{Lemma\ 6}$

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 $H \neq \emptyset$.

Lemma 7 $B \neq \emptyset$.

Proof The proof consists in showing that $H \subseteq B$. Then, as $H \neq \emptyset$, 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 17). Such a suspicion message can be sent by a process p_j only at Line 05 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 13, 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 $\leq nb[i, j]$, 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.
  - There is a time τ after which p_i continuously executes the while loop (Lines 01-03) 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 04).

    Let τ', $\tau' \geq \tau$, 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 $\geq hbc_i$ is ever sent, it follows that p_j processes all these heartbeat messages, i.e., it executes Line 13 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 05). 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 03), and heartbeat messages of consecutive batches carry increasing hbc_k values (Line 02). 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 04). 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 05) only because, since the last heartbeat message from p_i that entailed the setting of $timer_j[i]$ (at Line 13), p_j has not yet received a heartbeat message (to reset the timer, Line 13), or a stop_leader message (to stop the timer, Line 16) 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 05). 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 a 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.

$\Box_{Lemma\ 7}$

Let $(M_\ell, \ell) = lex_min(\{(M_i, i) \mid i \in B\})$.

Lemma 8 *There is a single process p_ℓ. Moreover p_ℓ is a correct process.*

Proof The lemma follows directly from the following observations: B does not contain faulty processes (definition), $B \neq \emptyset$ (Lemma 7), and no two processes have the same id (initial assumption). $\Box_{Lemma\ 8}$

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

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 06 or 16. 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 $sl_k \geq M_\ell$ if $i \notin B$, and p_j updates accordingly $susp_level_j[i]$ at Line 11. $\Box_{Lemma\ 9}$

Lemma 10 *There is a time after which p_ℓ executes forever the while loop of its Task $T1$ (Lines 01-03).*

Proof For each faulty process p_j, there is a finite time after which the predicate $j \notin contenders_\ell$ remains forever true (Lemma 6). For each correct process p_j, there is a finite time after which $j \notin contenders_\ell$ is always true, or $susp_level_\ell[j] \geq M_\ell$ (this follows from Lemma 9 and the fact that $M_\ell \leq M_j$, $\forall j \in B$). 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). $\Box_{Lemma\ 10}$

Theorem 3 *The protocol described in Figure 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 Lemma 8, 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 $susp_level_i[j] \geq M_\ell$. 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 03), 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 05) and the associated suppression of ℓ from $contenders_i$ (Line 06), 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 13). 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 06), a suspicion message is sent to p_ℓ (Line 05). 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.*[5] $\Box_{Theorem\ 3}$

Remark. The reader can verify that the Lines 14-16 of the protocol described in Figure 2 could be rewritten as follows:

if $(tag_k = stop_leader)$
  then $last_stop_leader_i[k] \leftarrow \max(last_stop_leader_i[k], hbc_k)$;
    stop $timer_i[k]$; $contenders_i \leftarrow contenders_i \setminus \{k\}$
endif.

This is because the reception by p_i of a stop_leader message sent by p_k can never entail the sending of a suspicion message from p_i to p_k.

5.3 Protocol Optimality

Theorem 4 *There is a time after which a single 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. $\Box_{Theorem\ 4}$

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

[5] 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.

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. $\Box_{Theorem\ 5}$

6 Conclusion

This paper has investigated the 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 the 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 $t-f$ 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 output links to the other correct processes are eventually timely. The lower bound result shows that this requirement is a necessary requirement. Interestingly, the second protocol enjoys another noteworthy property, namely, each value carried by a message is from a finite domain.

References

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

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

[3] Anceaume E., Fernández A., Mostefaoui A., Neiger G. and Raynal M., Necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors. *Journal of Computer and System Sciences,* 68:123-133, 2004.

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

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

[6] Chu F., Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.

[7] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[8] Guerraoui R., Indulgent Algorithms. *19th ACM Symposium on Principles of Distributed Computing, (PODC'00)*, ACM Press, pp. 289-298, 2000.

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

[10] Jiménez E., Arévalo S. and Fernández A., Implementing Unreliable failure Detectors with Unknown Membership. Submitted to *Information Processing Letters*, 2005.

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

[12] Lamport L., Shostak R. and Pease L., The Byzantine General Problem. *ACM Transactions on programming Languages and Systems*, 4(3):382-401, 1982.

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

[14] 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.

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

[16] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., From $\diamond\mathcal{W}$ to Ω: a Simple Bounded Quiescent Reliable Broadcast-based Transformation. *Tech Report #1759*, 7 pages, IRISA, University of Rennes 1 (France), 2005.

[17] Mostefaoui A. and Raynal M., Low-Cost Consensus-Based Atomic Broadcast. *7th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'2000)*, IEEE Computer Society Press, pp. 45-52, 2000.

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

[19] Mostefaoui A., Raynal M. and Travers C., Crash-resilient Time-free Eventual Leadership. *Proc. 23th Int'l IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 208-217, Florianopolis (Brasil), 2004.

[20] 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, pp. 109-118, Orlando (Florida), 2005.

[21] Pedone F. and Schiper A., Handling Message Semantics with Generic Broadcast Protocols. *Distributed Computing*, 15(2):97-107, 2002.

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

[23] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th Int. ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 297-308, Puerto Vallarta (Mexico), July 1998.