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

Antonio FERNÁNDEZ[†] Ernesto JIMÉNEZ[‡] Michel RAYNAL[⋆]

[†] LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain

[‡] EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain

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

anto@gsyc.escet.urjc.es ernes@eui.upm.es raynal@irisa.fr

## Abstract

*This paper considers the 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 are presented. The first protocol assumes that a process also knows the lower bound $\alpha$ 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 $\alpha$, 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. This protocol enjoys also the property that each message is made up of several fields, each of which taking values from a finite domain.*

## 1 Introduction

**Leader oracle: motivation** Failure detectors [4, 23] are at the core of a lot of fault-tolerant protocols encountered in asynchronous distributed systems. Among them, the class of *leader* failure detectors is one of the most important. This class, usually denoted $\Omega$, is also called the class of leader oracles. When clear from the context, the notation $\Omega$ will be used to denote either the oracle/failure detector class or an oracle of that class. An $\Omega$ 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 $\Omega$ has several noteworthy features. A fundamental one lies in the fact that, despite its very weak definition, it is powerful enough to allow solutions to 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 that problem to be solved (assuming a majority of correct processes) [5][1]. The liveness property of the well-known Paxos algorithm is based on such a leader oracle [12]. Other leader-based consensus protocols can be found in [10, 19].

Another major feature of $\Omega$ lies in the fact that it allows the design of *indulgent* protocols [9]. 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, they are correct. Moreover, $P$ always produces outputs when

[1] Let us remind that, while consensus can be solved in synchronous systems despite Byzantine failures of less than one third of the processes [13], it cannot be solved in asynchronous distributed systems prone to even a single process crash [8].

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, $\Omega$ is a class of oracles that allows designing indulgent protocols [9, 10].

Unfortunately, $\Omega$ cannot be implemented in pure asynchronous distributed systems where processes can crash. (Such an implementation would contradict the impossibility of solving consensus in such systems [8]. A direct proof of the impossibility to implement $\Omega$ in pure crash-prone asynchronous systems can be found in [20].) But thanks to indulgence, this is not totally bad news. More precisely, as $\Omega$ makes it possible the design of indulgent protocols, it is interesting to design "approximate" protocols that do their best to implement $\Omega$ 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 $\Omega$-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 $\Omega$-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 $\Omega$-based protocol can benefit from that period to produce an output.

The 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 $\Omega$ to be implemented during the "long periods" where they are satisfied.

**Related work** The very first implementations of $\Omega$ 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 $\tau_0$ after which there is a bound $\delta$ -possibly unknown- such that, for any time $\tau \geq \tau_0$, a message sent at time $\tau$ is received by time $\tau + \delta$) [14].

This "eventually timely links" approach has been refined to obtain weaker constraints. It has been shown in [1] that it is possible to implement $\Omega$ in a system where communication links are unidirectional, asynchronous and lossy, provided there is a correct process whose output links are eventually timely. The corresponding protocol implementing $\Omega$ 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 $\Omega$ 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 $\Omega$. 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 [15] where the notion of "eventual $t$-accessibility" is introduced. A process $p$ is $t$-accessible at some time $\tau$ if there is a set $Q$ of $t$ processes $q$ such that a message broadcast by $p$ at $\tau$ receives a response from all the processes of $Q$ by time $\tau + \delta$ (where $\delta$ 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 $\Omega$ when there is a process that is eventually $t$-accessible forever, and all other links are fair-lossy is described in [15].

A protocol based on a totally different approach to build $\Omega$ is described in [20]. It uses the time-free assumption proposed and investigated in [16]. 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 $\Omega$ to be implemented.

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* response 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 [20] that $\Omega$ 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 [16]. This approach has been extended to dynamic systems in [21] (systems where processes can dynamically enter or leave the system).

Another approach to build $\Omega$ is the "reduction" approach. This is a theoretical approach whose aim is to build $\Omega$ from other failure detector classes. Let us consider the failure detector class $\Diamond\mathcal{S}_x$ introduced in [3, 24]. 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 $\Omega$ in crash-prone asynchronous distributed systems equipped with a failure detector of the class $\Diamond\mathcal{S}$ [5, 6, 17]. A stacking of the previous protocols provides a protocol building $\Omega$ in an asynchronous system equipped with $\Diamond\mathcal{S}_{t+1}$ (this is a "reduction" of $\Omega$ to $\Diamond\mathcal{S}_{t+1}$).

As indicated, this approach is mainly theoretical: its aims is to investigate, compare and rank the computability power of failure detector classes. One of its most important results is the fact that the classes $\Omega$ 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, 17].

**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 [11] that this assumption is a necessary requirement for the classes $\Omega$ and $\Diamond\mathcal{S}$ to be equivalent. Actually, $\Diamond\mathcal{S}$ cannot be 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 $\Omega$ more attractive than $\Diamond\mathcal{S}$ as its implementation can require weaker assumptions. This paper investigates the implementation of $\Omega$ in asynchronous systems that satisfy rather weak assumptions on the initial knowledge of each process, and the behavior of the underlying network. Two protocols are presented.[2].

[2]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.

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 $\alpha$) on the number of correct processes. This means that all but $\alpha$ processes can crash in any run $R$ ($\alpha$ 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 it is 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.

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).

- (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, 11].

To our knowledge, [11] 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 $\alpha$, 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').

Interestingly, it is possible to state a lower bound on what can be done in an asynchronous system where the initial knowledge of any process includes neither $t$ nor $\alpha$. This lower bound states that, in such systems, there is no leader protocol in the runs where less than $n-1$ links eventually behave in a timely manner. Due to page limitation, the reader will find the proof of this theorem in [7].

**Roadmap** The paper is made up of four sections. Section 2 presents the distributed system model. Section 3 presents the first protocol and proves it is correct. Section 4 presents the communication-efficient protocol. Due to page limitation it has not been possible to include the proofs of these protocols (although they are very important). The reader can find them in [7].

# 2 Distributed System Model

## 2.1 Synchronous Processes with Crash Failures

The system is made up of a finite set $\Pi$ 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 $\tau_0$ and a bound $\delta$ such that each message sent by $p$ to $q$ at any time $\tau \geq \tau_0$ is received by $q$ by time $\tau + \delta$ ($\tau$ and $\delta$ 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 $\mu$, assuming that $p$ sends to $q$ infinitely many messages of the type $\mu$, $q$ (if it is correct) receives infinitely many messages of type $\mu$ 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.

**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 $\Omega$ of Oracles

$\Omega$ 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 $\tau$ and a correct process $p$ such that any invocation of *leader*() issued after $\tau$ returns $p$.

$\Omega$-based consensus algorithms are described in [10, 12, 19] 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, 12, 18, 22]).

## 3 A Leader Election Protocol

Assuming that each process knows its identity (K1), the lower bound $\alpha$ 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 (C1) and (C2). The proposed protocol tolerates message duplication. Finally, as far as the definition of *fair lossy link* is concerned, all the messages 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.

- $members_i$: set containing all the process ids that $p_i$ is aware of.
- $timer_i[j]$: 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 $suspected\_by_i[j]$: 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 $\alpha$, $p_i$ increases $susp\_level_i[j]$ and resets $suspected\_by_i[j]$ to $\emptyset$ for a new observation period.

- $sn_i$: local counter used to generate the increasing sequence numbers attached to each message sent by $p_i$.
- $state_i$: 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 $\eta$ 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 $state\_msg$.

### 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** [7] *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 $sn' \geq sn$.*

**Lemma 2** [7] *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.*

**Lemma 3** [7] *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 $SL_i$).*

**Lemma 4** [7] *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 $SL_i$).*

**Lemma 5** [7] *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.*

**Theorem 1** [7] *The protocol described in Figure 1 ensures that, after some finite time, all the correct processes have forever the same correct leader.*

## 4 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'). Moreover, it is assumed that there is no message duplication.

**Init**: allocate $susp\_level_i[i]$ and $suspected\_by_i[i]$; $susp\_level_i[i] \leftarrow 0$; $suspected\_by_i[i] \leftarrow \emptyset$;
$members_i \leftarrow \{i\}$; $to\_reset_i \leftarrow \emptyset$; $silent_i \leftarrow \emptyset$; $sn_i \leftarrow 0$;
$state_i \leftarrow \{(i, sn_i, \{(susp\_level_i[i], i)\}, silent_i)\}$ % initial knowledge (K1) %

---

**Task** $T1$:
**repeat forever every** $\eta$ **time units**
(01) $sn_i \leftarrow sn_i + 1$;
(02) **for each** $j \in silent_i$ **do** $suspected\_by_i[j] \leftarrow suspected\_by_i[j] \cup \{i\}$ **end for**;
(03) **for each** $j \in members_i$ **such that** $|suspected\_by_i[j]| \geq \alpha$ **do** % initial knowledge (K2) %
(04) $susp\_level_i[j] \leftarrow susp\_level_i[j] + 1$; $suspected\_by_i[j] \leftarrow \emptyset$ **end for**;
(05) replace $(i, -, -, -)$ in $state_i$ by $(i, sn_i, \{(susp\_level_i[j], j) \mid j \in members_i\}, silent_i)$;
(06) broadcast $(state_i)$;
(07) **for each** $j \in to\_reset_i$ **do** set $timer_i[j]$ to $timeout_i[j]$ **end for**; $to\_reset_i \leftarrow \emptyset$
**end repeat**

---

**Task** $T2$:
**when** *leader*() **is invoked by the upper layer**:
return $\big(\ell$ **such that** $(-, \ell) = lex\_min(\{(susp\_level_i[j], j)\}_{j \in members_i})\big)$

**when** $timer_i[j]$ **expires**:
(08) $timeout_i[j] \leftarrow timeout_i[j] + 1$; $silent_i \leftarrow silent_i \cup \{j\}$

**when** $state\_msg$ **is received**:
(09) **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 $sn' \geq sn\_k\ \}$;
(10) **for each** $(k, sn\_k, cand\_k, silent\_k) \in K$ **do**
(11) **if** $k \in members_i$ **then** replace $(k, -, -, -)$ in $state_i$ by $(k, sn\_k, cand\_k, silent\_k)$;
(12) stop $timer_i[k]$; $to\_reset_i \leftarrow to\_reset_i \cup \{k\}$; $silent_i \leftarrow silent_i \setminus \{k\}$
(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) $susp\_level_i[k] \leftarrow 0$; $suspected\_by_i[k] \leftarrow \emptyset$; $timeout_i[k] \leftarrow \eta$;
(16) $members_i \leftarrow members_i \cup \{k\}$; $to\_reset_i \leftarrow to\_reset_i \cup \{k\}$
**end if**
**end for**;
(17) **for each** $(k, sn\_k, cand\_k, silent\_k) \in K$ **do**
(18) **for each** $(sl, \ell) \in cand\_k$ **do** $susp\_level_i[\ell] \leftarrow \max(susp\_level_i[\ell], sl)$ **end for**;
(19) **for each** $\ell \in silent\_k$ **do** $suspected\_by_i[\ell] \leftarrow suspected\_by_i[\ell] \cup \{k\}$ **end for**
**end for**

**Figure 1. An eventual leader protocol (code for $p_i$)**

### 4.1 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 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 *leader*() = $i$ becomes true, and finishes when thereafter it becomes false (Lines 01-04).
- The counter $last\_stop\_leader_i[k]$ contains the great-

est $hbc_k$ 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 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 05)[3], 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 $susp\_level_i[i]$.

### 4.2 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 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 6** [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 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

[3]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.

```
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 $p_i$)**

$contenders_i$. 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.

$\Box_{Lemma\ 6}$

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** [7] $B \neq \emptyset$.

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

**Lemma 8** [7] *There is a single process $p_\ell$. Moreover $p_\ell$ is a correct process.*

**Lemma 9** [7] *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) $(i \in B \Rightarrow susp\_level_j[i] = M_i) \wedge (i \notin B \Rightarrow susp\_level_j[i] \geq M_\ell)$.*

**Lemma 10** [7] *There is a time after which $p_\ell$ executes forever the while loop of its Task $T1$ (Lines 01-03).*

**Theorem 2** [7] *The protocol described in Figure 2 ensures that, after some finite time, all the correct processes have forever the same correct process $p_\ell$ as common leader.*

## 4.3 Protocol Optimality

**Theorem 3** [7] *There is a time after which a single process sends messages forever.*

**Theorem 4** [7] *In an infinite execution, both the local memory of each process and the size of each message remain finite.*

## 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 $\Omega$ to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.

[7] Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony. *Tech Report #1770*, IRISA, Université de Rennes (France), 19 pages 2005.
http://www.irisa.fr/bibli/publi/pi/2005/1770.0770.html

[8] 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.

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

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

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

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

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 20.0.

[20] 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.

[21] 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.

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

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

[24] 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.