Specifying and Implementing an Eventual Leader Service for Dynamic Systems

Mikel Larrea, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1962 – Décembre 2010

Specifying and Implementing an Eventual Leader Service for Dynamic Systems

Mikel Larrea* Michel Raynal**

Abstract: The election of an eventual leader in an asynchronous system prone to process crashes is an important problem of fault-tolerant distributed computing. This problem is known as the implementation of the failure detector Ω. Nearly all papers that propose algorithms implementing such an eventual leader service consider a static system. In contrast this paper considers a dynamic system, i.e., a system in which processes can enter and leave. The paper has three contributions. It first proposes a specification of Ω suited to dynamic systems. Then, it presents and proves correct an algorithm implementing this specification. Finally, the paper discusses the notion of an eventual leader suited to dynamic systems. It introduces an additional property related to system stability. The design of an algorithm satisfying this last property remains an open challenging problem.

Key-words: Asynchronous system, Distributed algorithm, Dynamic system, Eventual leader, Failure detector, Fault-tolerance, Omega, Partial synchrony, Process crash, System model, Timely link.

Spécifier et implémenter un service de leadership pour les systèmes dynamiques

Résumé : *Ce rapport étudie la spécification et la mise en œuvre d'un service de ledaeship adapté aux systèmes dynamiques.*

Mots clés : *Leader inéluctable, système asynchrone dynamique, tolérance aux fautes.*

---

* University of the Basque Country, UPV/EHU, 20018 San Sebastián, Spain, mikel.larrea@ehu.es
** Senior Member Institut Universitaire de France, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France raynal@irisa.fr

1 Introduction

What is the eventual leader service Ω and why it is important Electing an eventual leader is a basic service for applications built on top of asynchronous crash-prone distributed systems. In the failure detector parlance [6, 24], such a service is called a failure detector of the class Ω [7]. Given a run, let a correct process be a process that does not crash in that run. The Ω service eventually provides the correct processes with a single leader that is a correct process.

The properties offered by such a service are pretty weak: a correct leader is eventually elected, but (1) there is no knowledge on when it is elected and (2) several (correct or not) leaders can coexist before a single correct leader is elected. Despite these very weak guarantees, Ω is a fundamental distributed computing service. This is due to several reasons among which the two following ones are noteworthy.

First Ω captures the weakest assumptions on process failures that allows the consensus problem to be solved. Consensus is one of the most important problems encountered in fault-tolerant asynchronous distributed computing. Its statement is very simple: assuming that each process proposes a value, all correct processes have to agree on the same value that has to be one of the proposed values [6, 12]. As an example of the use of consensus, let us consider the atomic broadcast problem that requires processes to deliver messages in the same order: consensus instances are used to agree on message delivery order [7, 25].

Another property of Ω is the fact that Ω-based algorithms are indulgent [13, 14]. A failure detector-based distributed algorithm is indulgent if, whatever the behavior of its underlying failure detector (Ω in our case) it never produces incorrect outputs. More precisely, if Ω behaves correctly, the algorithm produces outputs and those are correct. If Ω does not behave correctly (i.e., its behavior does not correspond to its specification) then the algorithm might not terminate but, if it does, its outputs are correct (more development can be found in [13, 25]).

Building Ω in static crash-prone asynchronous systems Ω cannot be implemented in pure asynchronous distributed systems prone to process crashes. If it could, consensus could be solved in such systems, which is known to be impossible [12]. A direct impossibility proof can be found in [25]. It follows that any asynchronous system has to be enriched with specific assumptions in order Ω can be built. Two main approaches have been investigated to implement Ω.

The first considers that the asynchronous system satisfies additional synchrony assumptions, namely, some links are eventually synchronous. According to the maximal number of processes that can crash, the number of eventually synchronous links, their structure, the fact they can change with time, etc., different assumptions have been proposed and algorithms based on these assumptions have been designed. Examples of such assumptions and algorithms can be found in [3, 4, 5, 6, 9, 15, 16, 17].

The second approach does not rely on timing assumptions and timeouts. It states a property on the message exchange pattern that, when satisfied, allows Ω to be implemented [21, 22]. The statement of such a property involves the maximal number of processes that can crash. Very recently, a very general approach combining and generalizing both previous ones has been proposed in [11]. It is important to notice that finding the weakest assumptions that asynchronous system behaviors have to satisfy in order Ω can be implemented is still an open problem.

The reader interested in the implementation of Ω in a shared memory distributed system will find appropriate assumptions and a corresponding algorithm in [10].

Building Ω in static crash-recovery asynchronous systems In a crash-recovery system, a crashed process may recover. Several assumptions to build an eventual leader Ω in such systems, and associated algorithms, have been proposed [18, 19]. It is assumed that there is a set of processes that, after some finite time, no longer crashes; the eventual leader is then one of them.

Building Ω in dynamic asynchronous systems A dynamic system is characterized by the fact that processes can enter and leave the system. The first (to our knowledge) algorithm implementing Ω in such systems is the one described is [23]. This algorithm is based on a general methodology that allows transforming a family of algorithms for static systems into algorithms for dynamic systems. An algorithm implementing Ω in the infinite arrival with bounded concurrency system model is described in [26]. Both previous algorithms are based on the assumption that, after some time, some processes do not crash and remain forever in the system.

Motivation and content of the paper This paper is motivated by the advent of new classes of systems and technologies such as VANETs, airborne networks, smart environments, broad area super-computing, cloud computing, etc., in which processes can enter and leave the system at arbitrary times. It is important to notice that, if no process remains long enough in such a system, nothing can be computed.

To address the kind of problems one has to face in such environments, the paper considers the implementation of Ω as a paradigm for real dynamic systems, i.e., systems where any process that entered the system can leave it at any time. Considering this line of research, the paper has three main contributions.

- The first is the specification of an eventual leader service (Ω) suited to real dynamic systems (as informally defined previously).
- The second contribution is an algorithm that, based on an appropriate synchrony assumption, implements the previous specification.

- The last contribution is a discussion on the nature of an eventual leader in a dynamic system. To that end, the paper proposes to enrich the specification of the eventual leader service with an additional stability property. Unfortunately (despite our efforts) implementing such a property remains an open problem (that could fortunately be solved by other researchers).

The proposed specification considers that if, from some point in time, the system behaves monotonically (either no process leaves the system or crashes, or no more processes join the system), then a common leader has to be eventually elected. The additional stability property considers that, whatever the system behavior, once a leader has been elected it is not demoted by other processes until it leaves the system or crashes (this property has been addressed in non-dynamic systems under the name "leader stability" [2, 8]).

Roadmap The paper is made up of 5 sections. Section 2 presents the dynamic asynchronous system model. Section 3 presents the definition of an eventual leader service, denoted $\Delta\Omega$, suited to dynamic systems. Then Section 4 presents a surprisingly simple distributed algorithm that solves the dynamic leader election problem when eventually the system satisfies some synchrony assumptions. Finally, Section 5 discusses the previous specification and proposes to enrich it with an additional stability property.

2 Dynamic system model at the application level

Processes and communication Intuitively an asynchronous dynamic system is a system that processes can enter and leave at arbitrary times. Moreover, a process can crash. We assume that the system is made up of an arbitrary number of processes. Each process has an identity and no two processes present in the system have the same identity. Identities are assumed to be totally ordered. We consider that i is the identity of the process denoted p_i. Processes are asynchronous in the following sense: while local statements are assumed to have a zero duration, there is no upper bound on the time taken by a process between any two consecutive communication events it issues.

The processes that are currently in the system can send and receive messages. To send a message, processes use an underlying broadcast facility that provides them with an operation denoted $\mathsf{broadcast}(m)$ where m is a message. This operation allows the invoking process to send a message. Message transfer delays are asynchronous which means that, while each message takes a finite duration, there is no upper bound on message transfer delays.

External time We also assume that there is a sequential time domain $\mathcal{T}$. This time, that is not known by the processes, can be seen as the time of an omniscient external observer. It is used only to describe the behavior of the system, specify the eventual leadership problem and prove correctness of associated algorithms. Without loss of generality we assume that no two communication events (broadcast or reception of messages) occur at the same time.

System dynamicity Let $\tau \in \mathcal{T}$ and τ_0 be the time at which the system starts. $\Pi(\tau)$ denotes the set of processes that compose the system at time τ. Hence, $\Pi(\tau_0) \neq \emptyset$ is the initial set of processes.

Let us consider a process p_i that joins the system. This occurs at some time denoted $\tau(join_i)$. Similarly, if it ever leaves the system, its departure time is denoted $\tau(leave_i)$. If process p_i, that has entered the system and not yet left, crashes, $\tau(crash_i)$ denotes the time at which it crashes.

If a process p_i that joined the system leaves it or crashes $\tau(exit_i)$ denotes $\tau(leave_i)$ (if p_i leaves the system) or $\tau(crash_i)$ (if it crashes). If p_i neither leaves the system nor crashes, $\tau(exit_i) = +\infty$.

It is assumed that, at any time, there is at least one process in the system: $\forall \tau : \Pi(\tau) \neq \emptyset$. Moreover, a process enters the system at most once. This means that a process that wants to re-enter the system has to appear as a new process. To that end it is sufficient for it to use a new identity.

Notation The previous dynamic system model is denoted $\Delta\mathcal{AS}[\emptyset]$. ($\Delta$ is for "dynamic" and $\mathcal{AS}$ is for "asynchronous system". $[\emptyset]$ is used to indicate that the system is not enriched with additional assumptions.)

3 Eventual leader election in a dynamic system: a specification

The intuition that underlies the definition The specification of the eventual leader election problem in an asynchronous dynamic system is made up of the following properties.

The first property, denoted EL_NI (for Eventual Leadership in Non-Increasing systems), states that, if after some time the system does not increase, a leader must eventually be elected. This property is the classical eventual leadership property that defines the eventual leader failure detector class Ω in non-dynamic asynchronous systems [7].

The second property, denoted EL_ND (for Eventual Leadership in Non-Decreasing systems), is similar to the previous property except that it considers the case where after some time the system does not decrease. It states that, in such system behaviors, (1) a leader

must eventually be elected and (2) the new processes that join the system after a leader has been elected have to eventually consider this process as their leader.

The local variable $leader_i$ of process p_i Each process p_i has a local variable denoted $leader_i$ whose aim is to contain the identity of the leader. When $p_i \in \Pi(\tau)$, the notation $leader_i^\tau$ is used to denote the value of $leader_i$ at time τ. Initially, we have $leader_i^{\tau(join_i)} = \perp$, i.e., when p_i joins the system it has no leader. The aim of a leader election algorithm is to give a common and correct non-$\perp$ value to these local variables.

Formal statement of the EL_NI property If after some time τ, no process enters the system, a leader is eventually elected.

$$\big[\exists\, \tau : \forall\, \tau_1, \tau_2 \geq \tau : (\tau_1 \geq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2))\big] \\ \Rightarrow \big[\exists\, p_\ell \in \textstyle\bigcap_{\tau' \geq \tau} \Pi(\tau') : \exists\, \tau_1 \geq \tau : \forall\, \tau_2 \geq \tau_1 : \forall\, p_i \in \Pi(\tau_2) : leader_i^{\tau_2} = \ell\big].$$

Formal statement of the EL_ND property While EL_NI addressed the case where, after some time, the system only decreases, the property EL_ND is its counterpart for the case where after some time, the system only increases. If after some time τ, no process leaves the system or crashes, a leader has to be eventually elected and the new processes eventually adopt it.

$$\big[\exists \tau :\ \forall\, \tau_1, \tau_2 \geq \tau : (\tau_1 \leq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2))\big] \\ \Rightarrow \big[\exists \tau_1 \geq \tau, \exists\, p_\ell \in \Pi(\tau_1) : \\ \text{(i)} \quad \big[\forall\, \tau_2 \geq \tau_1 : \forall\, p_i \in \Pi(\tau) : leader_i^{\tau_2} = \ell\big] \\ \text{(ii)} \ \wedge \big[\forall\, \tau' \geq \tau : \big(p_i \in \Pi(\tau') \setminus \Pi(\tau)\big) \Rightarrow \big(\exists\, \tau_0 \geq \tau(join_i) : \forall\, \tau'' \geq \tau_0 : leader_i^{\tau''} = \ell\big)\big] \\ \big].$$

Item (i) states that all processes that are in the system when it stops decreasing (this defines time τ) eventually agree on a common leader p_ℓ. Item (ii) addresses the case of a process that joins the system after time τ. Eventually its variable $leader_i$ has to contain ℓ forever.

The class $\Delta\Omega$ of failure detectors This class, suited to dynamic systems, includes all failure detectors that, at any time, provide the processes p_i currently present in the system with local variables $leader_i$ that collectively satisfy the properties EL_NI and EL_ND.

The price that has to be paid to face dynamicity It is important to remark that if processes enter and leave the system in such a way that no process remains "long enough" in the system, then it is possible that no common process can ever be elected. The previous definition takes this case into account. Said differently, a (possibly temporary) leader is required to be elected only when the system "behaves monotonically" (either no more arrivals or no more departures) during "long enough" periods of time. This is captured by the properties EL_NI and EL_ND, respectively.

4 Implementing an eventual leader in a dynamic system

This section proposes an algorithm implementing a failure detector of the class $\Delta\Omega$ defined in the previous section, i.e., satisfying the properties EL_NI and EL_ND. Interestingly, as the algorithm presented in [18], the proposed algorithm is communication-efficient. This means that, if the system behaves monotonically (after some time either it does no longer increase or does no longer decrease), then eventually only the elected leader keeps on sending messages.

4.1 Enriching the system with additional assumptions

It is known that Ω cannot be implemented in a static asynchronous crash-prone distributed system [7, 25]. Hence, in order to implement $\Delta\Omega$ in a dynamic system we make the following eventual synchrony assumptions denoted $\diamond ES$. The system model $\Delta\mathcal{AS}[\emptyset]$ enriched with the behavioral assumptions $\diamond ES$ is denoted $\Delta\mathcal{AS}[\diamond ES]$.

- Processing times are considered as negligible with respect to message transfer delays.
- Each process has a local clock that can measure time intervals with a bounded drift (unknown by the processes). These clocks are not required to be synchronized. Each clock progresses forever. We assume that the current value of the local clock of a process entering the system is as large as the clock value of the processes already in the system (in practice, this can be easily ensured by accessing a GPS-like system or using NTP).
- Communication links cannot create or alter messages. Concerning timeliness or loss properties, we assume that the communication is eventually timely [4] among the processes present in the system. Eventual timeliness means that there is a bound δ on message transfer delays and a time, denoted GST and called *global stabilization time*, such that any message m broadcast at a time $\tau \geq GST$ is received by time $\tau + \delta$ at every process that is present in the system from time τ to time $\tau + \delta$.

4.2 An eventual leader algorithm

The idea is to elect the process that is present in the system since the longest time period, i.e., the process that has joined the system $first$ and has neither left it nor crashed yet. To do so, we use the local clocks to associate a timestamp with each process join (process identities are used -if needed- to select a process among those that have joined the system with the same clock value). As we are about to see, a noteworthy property of Algorithm 1 is its simplicity (when compared to other eventual leader election algorithms).

Local variables To read its local clock, a process p_i invokes the operation $\mathsf{clock}_i()$. It also manages the following local variables.

- $leader_i$ contains the identity of the process that is p_i's current leader, or $\perp$ if p_i has no current leader.
- τ_joined_i contains the local time of p_i's join, while $min\_\tau\_joined_i$ contains the time of the join of the process that p_i considers currently as leader (i.e., p_{leader_i}).
- $timeout_i$ is a local variable that is increased each time the timer expires.
- $timer_i$ is a local timer. When needed, it is set to the value contained in $timeout_i$.

```
when p_i joins the system:
(01)  leader_i ← ⊥; τ_joined_i ← clock_i();
(02)  min_τ_joined_i ← τ_joined_i; timeout_i ← τ_joined_i;
(03)  wait timeout_i time units;
(04)  if (leader_i = ⊥) then leader_i ← i
(05)                    else set timer_i to timeout_i end if;
(06)  repeat forever every β time units
(07)        if (leader_i = i) then broadcast (τ_joined_i, i) end if
(08)  end repeat.

when (τ_joined, j) is received such that j ≠ i:
(09)  if   (τ_joined < min_τ_joined_i)
(10)     ∨ ((τ_joined = min_τ_joined_i) ∧ (leader_i ≠ ⊥) ∧ (j ≤ leader_i))
(11)     ∨ ((τ_joined = min_τ_joined_i) ∧ (leader_i = ⊥) ∧ (j < i))
(12)        then leader_i ← j;
(13)             min_τ_joined_i ← τ_joined;
(14)             set timer_i to timeout_i
(15)  end if.

when timer_i expires:
(16)  timeout_i ← timeout_i + 1;
(17)  leader_i ← i;
(18)  min_τ_joined_i ← τ_joined_i.
```

Algorithm 1: Building $\Delta\Omega$ in $\Delta\mathcal{AS}[\diamond ES]$ (code for p_i)

Process behavior As already suggested, the aim of the algorithm is to elect the correct process p_ℓ whose timestamp (τ_joined_ℓ, ℓ) is the smallest. The behavior of a process p_i is consequently the following one.

- When a process p_i joins the system, it sets $leader_i$ to $\perp$, and τ_joined_i, $min\_\tau\_joined_i$ and $timeout_i$ to the current value of its local clock. Then, it waits for $timeout_i$ time units. This is an observation period during which it can receive messages that update its context and provides it a current leader (see next item). The initialization of $timeout_i$ to the current value of the local clock, combined with the fact that this clock always increases, prevents new joining processes from forever disturbing the leader election.

  When this observation period terminates, p_i considers it is leader if $leader_i$ has not been updated. Otherwise, it sets $timer_i$ to the value of $timeout_i$ in order to monitor the process that it considers as its current leader.

  Finally, p_i enters an infinite loop in which it regularly (every β time units) does the following: if it currently considers itself as leader, it broadcasts the message (τ_joined_i, i).
- When p_i receives (τ_joined, j) such that $j \neq i$, it does the following. If (a) $\tau\_joined < min\_\tau\_joined_i$ or (b) $leader_i \neq \perp$ and $(\tau\_joined, j) \leq (min\_\tau\_joined_i, leader_i)$, p_i demotes its current leader and replaces it by p_j (observe that (b) also covers the case where $leader_i$ was set to j, in which case p_i "reaffirms" p_j as its leader). If p_i has no leader and p_j is a better candidate than itself (i.e., $leader_i = \perp$ and $(\tau\_joined, j) < (min\_\tau\_joined_i, i)$), p_i considers p_j as its current leader.

  If p_j is considered as new leader, p_i updates accordingly $leader_i$ and $min\_\tau\_joined_i$, and sets $timer_i$ to the current value of $timeout_i$ in order to start a monitoring period with respect to p_j.

- When $timer_i$ expires, p_i suspects its current leader p_{leader_i} to have either left the system or crashed. It consequently increases $timeout_i$ and considers itself as the new leader. As indicated in the first item, this entails the periodical broadcast of (τ_joined_i, i) messages.

4.3 Correctness proof of the algorithm

Theorem 1 *Algorithm 1 implements a failure detector of the class $\Delta\Omega$ in $\Delta\mathcal{AS}[\diamond ES]$ (i.e., it satisfies the properties EL_NI and EL_ND). Moreover, it is communication-efficient.*

Proof *Proof of the EL_NI property.* This property addresses the case where after some time no new process joins the system. Let τ be such a time instant. Among the processes that remain forever in the system (remember that we assume that at any time there is at least one process in the system), let p_ℓ be the process with the smallest timestamp (τ_joined_ℓ, ℓ). We claim that eventually p_ℓ becomes the permanent leader for all the processes that remain forever in the system. This claim follows from the following observations.

1. Eventually all the processes having a smaller timestamp leave the system or crash. As a consequence, they stop broadcasting messages, and eventually all their messages disappear from the system.
2. Eventually process p_ℓ considers permanently itself as a leader. This is because, after some finite time, it does not receive any message with a timestamp smaller than (τ_joined_ℓ, ℓ). Consequently, p_ℓ starts broadcasting periodically a (τ_joined_ℓ, ℓ) message forever (Line 07).
3. Eventually the communication from p_ℓ to any other process p_i becomes timely. Moreover, every time $timer_i$ expires, p_i increases $timeout_i$ (Line 16). Consequently, any process p_i ($i \neq \ell$) may demote p_ℓ's leadership only a finite number of times, at the end of which it considers p_ℓ as its permanent leader.

Finally, as far as communication efficiency is concerned, it follows from the previous observations that eventually only p_ℓ sends messages.

Proof of the EL_ND property. Any time instant τ considered in this proof is assumed to be greater than the global stabilization time (i.e., $\tau > GST$). Also, for every process p_i joining the system, we assume that $\tau\_joined_i > \beta + \delta$. Observe that this eventually occurs by the behavioral assumption on the local clocks of processes.

The EL_ND property addresses the case where after some time no process leaves the system or crashes. Let τ be such a time instant. Among the processes that have joined the system by time τ and remain forever in the system, let p_ℓ be the process with the smallest timestamp (τ_joined_ℓ, ℓ). We claim that eventually p_ℓ becomes the permanent leader for all the processes that either (1) have joined the system by time τ and do not leave it or crash, or (2) join the system after time τ. The proof of this claim is similar to the previous one (for proving the EL_NI property). It follows from the following observations.

1. Eventually all the messages broadcast by processes having left the system or crashed before time τ disappear from the system.
2. Any process p_i that joins the system after time τ is such that $\tau\_joined_i > \tau\_joined_\ell$. Hence, in the competition between p_i and p_ℓ to become the leader, eventually p_ℓ is the winner.
3. Eventually process p_ℓ considers permanently itself as a leader. This is because, after some finite time, it does not receive any message with a timestamp smaller than (τ_joined_ℓ, ℓ). Consequently, p_ℓ starts broadcasting periodically a (τ_joined_ℓ, ℓ) message forever (Line 07).
4. Eventually the communication from p_ℓ to any other process p_i becomes timely. Moreover, every time $timer_i$ expires, p_i increases $timeout_i$ (Line 16). Consequently, any process p_i ($i \neq \ell$) may demote p_ℓ's leadership only a finite number of times, at the end of which it considers p_ℓ as its permanent leader.

Regarding communication efficiency, observe that eventually only p_ℓ sends messages among the processes that have joined the system by time τ. Moreover, by the assumption that $\tau\_joined_i > GST$ and $\tau\_joined_i > \beta + \delta$ for every process p_i joining the system after time τ, and since $timeout_i$ is initialized to τ_joined_i (Line 02) and never decreases in the algorithm, we can conclude that during its observation period (Line 03) and any subsequent monitoring period p_i receives a (τ_joined_ℓ, ℓ) message from p_ℓ. Hence, we have that no process joining the system after time τ sends messages. Consequently, eventually only p_ℓ sends messages, and the algorithm is communication-efficient. $\square_{Theorem\ 1}$

5 Discussion: On the specification of Ω for dynamic systems

Motivation The specification of the eventual leader service (Ω) for dynamic systems with the properties EL_NI and EL_ND can be questioned. This is because these properties consider only the cases where the dynamic system becomes monotonous in the sense that an eventual leader is required to be elected only if after some finite time the system either only "decreases" (property EL_NI) or only "increases" (property EL_ND). It is possible that the system never behaves that way, and, in that case, the specification leaves open the possibility that no common leader be ever elected. This observation motivates the definition of a stronger specification than EL_NI + EL_ND. This section proposes such a stronger specification.

A property to capture stable periods This property, denoted STAB, is related to the stability of the leader election. It states that, if there is a time τ at which a leader has been elected and the system is in a sanitized state, then this leader cannot be demoted until it crashes or leaves the system. We consider here that a system is in a *sanitized* state if it cannot be "polluted" by irrelevant (i.e., too old) messages. More precisely, the system is in a sanitized state at time τ if none of the messages currently in transit has been sent by a process that has crashed or left the system before time τ.

Preliminary definitions The formal definition of the STAB property is based on the following definitions.

- Given a message m, $m.sender$ denotes the identity of its sender.
- $MSG\_inT(\tau)$ denotes the set of messages that are in transit at time τ.
- $LEFT(\tau)$ is the set of processes that have entered the system and then left it or crashed by time τ.
- $SANE(\tau) \equiv \big[\forall m \in MSG\_inT(\tau) :\ m.sender \notin LEFT(\tau)\big]$.

Formal statement of the additional STAB property This property states that as soon as there is a time τ such that (1) all processes in $\Pi(\tau)$ agree on the same leader p_ℓ and (2) the system is sane (in the sense it cannot be polluted by old messages from processes that have left the system or have crashed) then any process p_i in $\Pi(\tau)$ continues to consider p_ℓ as its leader until p_i or p_ℓ exits the system by leaving or crashing (Item i). Moreover, if time permits (Item ii) all processes joining the system eventually consider p_ℓ as their leader. Let us observe that Item ii is not redundant with respect to the previous EL_NI and EL_ND properties, as it covers the case where p_ℓ remains forever in the system while processes are permanently entering and leaving the system.

$$
\begin{array}{l}
\big[\exists\, \tau,\ \exists p_\ell \in \Pi(\tau) : \big((\forall\, p_i \in \Pi(\tau) : leader_i^\tau = \ell) \wedge SANE(\tau)\big)\big] \\
\quad \Rightarrow \big[\forall\, \tau' :\ \tau \leq \tau' \leq \tau(exit_\ell) : \\
\qquad \text{(i)} \quad \big[\forall\, p_i \in \Pi(\tau) \cap \Pi(\tau') : leader_i^{\tau'} = \ell\big] \\
\qquad \text{(ii)} \ \wedge \big[\forall\, p_i \in \Pi(\tau') \setminus \Pi(\tau): \\
\qquad\qquad \big[(\tau(exit_i) = \tau(exit_\ell) = +\infty) \Rightarrow \big(\exists\, \tau'' \geq \tau(join_i) :\ \forall \tau''' \geq \tau'' :\ leader_i^{\tau'''} = \ell\big)\big] \\
\qquad\qquad \big] \\
\quad \big].
\end{array}
$$

To conclude: the state of affairs Despite our efforts, we have not yet succeeded in designing an algorithm that satisfies simultaneously the three properties EL_NI, EL_ND and STAB. Hence, the following questions: is STAB too strong to be implemented? If it is, how to weaken it ("as much as possible") to obtain an implementable specification? These questions are left as challenging research issues for future work.

Strengthening the system synchrony in order to satisfy STAB *vs* weakening the STAB property The difficulty in satisfying the STAB property comes from the fact that, even if all processes in the system agree on the same leader p_ℓ and the system is sane, p_ℓ can be demoted by a process p_i if $timeout_i$ is not properly set and expires prematurely.

In this regard, STAB can be satisfied in a synchronous system, where the message transmission delay is bounded and the bound is known. More precisely, let us assume that there is a known bound δ on message delay, such that any message m broadcast at a time τ is received by time $\tau + \delta$ at every process that is present in the system from time τ to time $\tau + \delta$. Let us also assume that the skew between any two clocks in the system is bounded by θ (with θ known by processes). Algorithm 1 satisfies STAB by doing the following simple modifications:

- Initializing $timeout_i$ to $\beta + \delta + 2\theta$ (Line 02).
- Removing the instruction that increases $timeout_i$ (Line 16).

Actually, in order to satisfy STAB it is sufficient that the communication *from the leader p_ℓ to the rest of processes* behaves timely.

Alternatively, we can weaken the STAB property, by redefining it as $\diamond$STAB (or Eventual_STAB). Basically, $\diamond$STAB states that the STAB property is satisfied eventually instead of "as soon as processes agree on a leader and the system is sane". Interestingly, Algorithm 1 satisfies $\diamond$STAB.

References

[1] Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT-NEWS*, 35(2):36-59, 2004.

[2] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Stable Leader Election. *Proc. 15th Int'l Symposium on Distributed Computing (DISC'01)*, Springer-Verlag LNCS #2180, pp. 108-122, 2001.

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

[4] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Omega in Systems with Weak Reliability and Synchrony Assumptions. *Distributed Computing*, 21(4):285-314, 2008.

[5] Biely M. and Widder J., Optimal Message-driven Implementations of Omega with Mute Processes. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1): Article 4, 22 pages, 2009.

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

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

[8] Delporte-Gallet C., Devismes S. and Fauconnier H., Robust Stabilizing Leader Election. *Proc. 9th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, Springer-Verlag LNCS #4838, pp. 219-233, 2007.

[9] Fernández A., Jiménez E. and Raynal M., Eventual Leader Election with Weak Assumptions on Initial Knowledge, Communication Reliability, and Synchrony. *Proc. Int'l IEEE conference on Dependable Systems and Networks (DSN'06)*, IEEE Society Press, pp. 166-175, 2006.

[10] Fernández A., Jiménez E., Raynal M. and Trédan G., A Timing Assumption and Two t-Resilient Protocols for Implementing an Eventual Leader Service in Asynchronous Shared Memory Systems. *Algorithmica*, 56(4): 550-576, 2010.

[11] Fernández A. and Raynal M., From an Asynchronous Intermittent Rotating Star to an Eventual Leader. *IEEE Transactions on Parallel and Distributed Systems*, 21(9): 1290-1303, 2010.

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

[13] Guerraoui R. and Lynch N.A., A General Characterization of Indulgence. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4): Article 20, 2008.

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

[15] Hutle M., Malkhi D., Schmid U. and Zhou L., Chasing the Weakest System Model for Implementing Ω and Consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4): 269-281, 2009.

[16] 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)*, pp. 52-60, 2000.

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

[18] Martín C. and Larrea M., A Simple and Communication-efficient Omega Algorithm in the Crash-recovery Failure Model. *Information Processing Letters*, 110(3):83-87, 2010.

[19] Martín C., Larrea M. and Jiménez E., Implementing the Omega Failure Detector in the Crash-recovery Failure Model. *Journal of Computer and System Sciences*, 75(3):178-189, 2009.

[20] Merritt M. and Taubenfeld G., Computing with Infinitely Many processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 164-178, 2000.

[21] Mostéfaoui A., Mourgaya E., Raynal M. and Travers C., A Time-free Assumption to Implement Eventual Leadership. *Parallel Processing letters*, 16(2):189-208, 2006.

[22] Mostéfaoui A., Raynal M. and Travers C., Time-free and Timer-based Assumptions can be Combined to Get Eventual Leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656-666, 2006.

[23] Mostéfaoui A., Raynal M., Travers C., Patterson S., Agrawal D. and El Abbadi A., From Static Distributed Systems to Dynamic Systems. *Proc. 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, IEEE Pres, pp. 109-118, 2005.

[24] Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopedia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).

[25] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Distributed Systems. *Morgan & Claypool Publishers*, 265 pages, 2010, ISBN 978-1-60845-293-4.

[26] Tucci-Piergiovanni S. and Baldoni R., Eventual Leader Election in Infinite Arrival Message-Passing System Model with Bounded Concurrency. *Proc. 8th European Dependable Computing Conference (EDCC'10)*, CPS, pp. 127-134, 2010.