Eventual Leader Service in Unreliable Asynchronous Systems: Why? How?

Michel RAYNAL
IRISA, Université de Rennes, Campus de Beaulieu 35 042 Rennes, France
raynal@irisa.fr

Abstract

Providing processes with an eventual leader service is an important issue when one has to design and implement a middleware layer on top of a failure-prone asynchronous distributed system. This invited lecture investigates this problem. It first shows that such a service cannot be built if the underlying system is fully asynchronous. Then, the paper visits several additional behavioral assumptions that have been proposed in the literature to cope with this impossibility and presents corresponding eventual leader election protocols. This lecture can be seen as a guided tour of the eventual leader service problem, whose aim is to benefit researchers and system engineers working in distributed middleware built on top of asynchronous networks.

Keywords*: Assumption coverage, Asynchronous system, Behavioral assumption, Distributed algorithm, Eventual t-source, Eventual leader, Failure detector, Fault-tolerance, Message pattern, Omega, Oracle, Partial synchrony, Process crash, System model, Eventual timely link.*

1 Introduction

The problem Implementing an eventual leader service (i.e., solving the *eventual leader election* problem) consists in providing the processes of a failure-prone asynchronous distributed system with a primitive denoted leader() (1) that returns a process identity (pid) each time it is called, and (2) after some finite time, always returns the same pid, that pid being the identity of a non-failed process. When one is interested in solving that problem, a part of the difficulty lies in the type of failures we want to cope with. This paper considers the simplest failure model, namely, the process crash failure model. This means that a process can halt prematurely (process crash failure) and, after it has crashed, a process executes no computation step (there is no recovery). There is no process Byzantine failure, and the communication medium used by the processes to communicate information and synchronize is assumed to be reliable.

A synchronous distributed system is characterized by known bounds on process speed and message delay. It is relatively easy to eventually elect a leader in such a system, despite process crashes. A simple protocol implementing an eventual leader service consists in directing the processes to progress by rounds (i.e., in a lock-step manner) in such a way that, at each round, every non-crashed process (1) sends a message to each other process, and (2) elects as its current leader the process with the smallest pid from which it has received a message during that round. Given a run of the system, it is easy to see that, after the last round during which a process crashes, all the non-crashed processes elect forever the same leader.

An asynchronous distributed system is characterized by the absence of bounds on process speed and message delay. In such a context, it becomes impossible for a process to distinguish a crashed process from a process that is very slow or a process with which communications are very slow. This impossibility makes some problems impossible to solve. Eventual leader election is such an impossible problem to solve in asynchronous distributed systems prone to process crash failures (ore more severe failures). This impossibility is inherently due to the net effect of asynchrony and failures.

Why an eventual leader service is important: on a more practical side The eventual leader election problem is a problem encountered in a lot of applications as shown by the two following examples.

It is well-recognized that active replication is a way to cope with failures. This is the well-known *state machine* replication paradigm. Clients send their commands (operations) to a "logical" server (state machine) that is physically replicated. Each alive replica executes the command, and sends back an answer to the client. Finally, the client considers the first answer it receives. The main issue for the replicas is to guarantee the one-copy semantics, i.e., from a logical point of view, everything has to appear as if there was a single reliable server that processes all the commands (operations). To obtain this goal, the servers have to coop-

erate in order to process the commands in the same order[1]. This is a typical agreement problem that has received a lot of attention, namely the *consensus* problem: the alive replicas have to agree on the order in which they have to process the commands issued by the clients. It has been shown that the output of an eventual leader service represents the weakest service (in terms of information on failures) required to solve the consensus problem [5]. So, the ability to implement a replicated state-machine relies on the ability to solve the consensus problem that, in turn, relies on an eventual leader service.

Let us consider a set of interconnected sensors, deployed on some area. The aim of this set of sensors is to repeatedly sense its environment and send the corresponding data to a base station. In order to save energy and prevent message clogging at the base station, it is required that eventually one sensor only senses the environment and sends the corresponding data. Then, when the selected sensor crashes (e.g., battery exhaustion), another sensor has to be selected, etc. As a sensor network is a crash-prone asynchronous distributed system, the problem we are faced with is a typical leader election problem: determine a unique sensor that, until it crashes, plays a particular role.

Why an eventual leader service is important: on a more theoretical side An eventual leader service is specified by a set of abstract properties. These properties define an object, usually denoted Ω [5], that belongs to the family of failure detector objects [4, 23]. The failure detector objects are also called *oracles*. The oracle Ω has a noteworthy feature, namely, it allows the design of *indulgent* protocols [10].

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 the underlying oracle meets its specification. The only case where P can be prevented from producing outputs is when the implementation of 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 an oracle that allows designing indulgent protocols [10, 11]. From a protocol design point of view, as it is not know when the eventual leader is elected, the main main work of an Ω-based protocol consists is guaranteeing that its safety property is never violated.

Content of the paper An eventual leader service cannot be implemented in a pure (shared memory or message-passing) asynchronous system as soon as even a single process can crash. So, considering asynchronous message-passing systems, the paper (1) focuses on additional assumptions (proposed so far) that, when satisfied by the underlying system, allow implementing an eventual leader, and (2) presents corresponding protocols. The paper first shows that no eventual leader service Ω can be implemented in a fully asynchronous system prone to process crashes (Section 3). It then quickly surveys the different approaches that have been proposed to circumvent this impossibility (Section 4). The paper then presents several of these approaches in more details (Section 5 to Section 8).

After having shown the difficulty of the problem, the aim is to give the readers an idea of the type of behavioral assumptions that allows electing an eventual leader. A more ambitious goal of this lecture is to provide the readers with concepts and techniques that allow obtaining a better view of the uncertainty that has to be mastered when one has to develop reliable distributed middleware.

[1] Assuming (to simplify) that the server provides the clients with deterministic commands.

2 A simple base model for asynchronous message-passing systems

We consider systems consisting of a finite set Π of n processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. The integer i is the identity of p_i (pid). A process can fail by *crashing*, i.e., by prematurely halting. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. A process that does not crash in a run is *correct* in that run. Otherwise, it is *faulty* in that run. The model parameter t denotes the maximum number of processes that may crash in any run ($1 \leq t < n$).

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are assumed to be reliable: they do not create, alter or lose messages. In particular, if p_i sends a message to p_j, eventually p_j receives that message (unless p_j fails). (Let us observe that channels are not required to be FIFO.)

This base model is characterized by the fact that there are assumptions neither on the speed of one process with respect to another, nor on message delays. This is the classical *asynchronous* message-passing system made up n processes where up to t may crash. It is denoted $AS_{n,t}[\emptyset]$ in the following. To simplify the presentation we consider in the rest of the paper that a local processing takes no time.

3 Impossibility of electing an eventual leader in a pure asynchronous system

Consensus can be solved in an asynchronous system with a majority of correct processes, equipped with an eventual

leader service [5, 11, 14, 19]. It consequently follows from the fact that the consensus problem cannot be solved in purely asynchronous systems [9], that an eventual leader service cannot be implemented in an asynchronous system $AS_{n,t}[\emptyset]$ with $1 \leq t < n/2$. The theorem that follows shows a more general result in the sense that it requires no constraint on t. (Its proof is a *direct* impossibility proof in the sense that it does not rely on the impossibility of solving another distributed computing problem -such as the consensus problem [9]-. It is inspired from [3].)

Theorem 1 *No eventual leader service can be implemented in* $AS_{n,t}[\emptyset]$ *with* $1 \leq t < n$.

Proof The proof is by contradiction. Assuming that there is a protocol implementing an eventual leader service, we construct a crash-free execution in which there is an infinite sequence of leaders such that any two consecutive leaders are different, from which it follows that the eventual leadership property cannot not satisfied.

- Let R_1 be a crash-free execution, and t_1 be the time after which some process p_{ℓ_1} is elected as the definitive leader.

  Moreover, let R'_1 be an execution identical to R_1 until $t_1 + 1$, and where p_{ℓ_1} crashes at $t_1 + 2$.

- Let R_2 be a crash-free execution identical to R'_1 until $t_1 + 1$, and where the messages sent by p_{ℓ_1} after $t_1 + 1$ are arbitrarily delayed (until some time that we will specify later).

  As, for any process $p_x \neq p_{\ell_1}$, R_2 cannot be distinguished from R'_1, it follows that some process $p_{\ell_2} \neq p_{\ell_1}$ is elected as the definitive leader at some time $t_2 > t_1$. After p_{ℓ_2} is elected, the messages from p_{ℓ_1} can be received.

  Moreover, let R'_2 be an execution identical to R_2 until $t_2 + 1$, and where p_{ℓ_2} crashes at $t_2 + 2$.

- Let R_3 be a crash-free execution identical to R'_2 until $t_2 + 1$, and where the messages from ℓ_2 are delayed (until some time that we will specify later).

  Some process $p_{\ell_3} \neq p_{\ell_2}$ is elected as the definitive leader at some time $t_3 > t_2 > t_1$. After p_{ℓ_3} is elected, the messages from p_{ℓ_2} are received. Etc.

This inductive process, repeated indefinitely, constructs a crash-free execution in which an infinity of leaders are elected at times $t_1 < t_2 < t_3 < \ldots$ and such that no two consecutive leaders are the same process. It follows that there is no finite time after which the same correct process is forever elected as the single common leader. $\square_{Theorem\ 1}$

4 Existing approaches to build an eventual leader service

Up to now two main approaches have been investigated to implement an eventual leader service (Ω) in crash-prone asynchronous distributed systems. Both enrich the asynchronous system with additional assumptions that, when satisfied, allow implementing Ω. These approaches are orthogonal: one is related to timing assumptions, the other is related to message pattern assumptions.

The eventual timely link approach The first approach considers that the asynchronous system eventually satisfies additional *synchrony* properties. Considering a reliable communication network, the very first papers (e.g., [15]) assumed that all the links are *eventually timely*. This assumption means that 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 $\tau + \delta$. A protocol based on such an assumption is presented in Section 5.

This approach has then been refined to obtain weaker and weaker assumptions. 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 that there is a correct process whose $n - 1$ output links are eventually timely. This assumption has further been weakened in [2] where it is shown that Ω can be built as soon as there is a correct process that has only t eventually timely links (let us recall that t is an upper bound on the number of processes that can crash in a run); such a process is called an *eventual t-source*. (Let us notice that, after the receiver has crashed, the link from a correct process to a crashed process is always timely). A protocol based on such a t-source assumption is presented in Section 6.

Another time-based assumption has been proposed in [16] where the notion of *eventual t-accessibility* is introduced. A process p is eventual t-accessible if there is a time τ_0 such that, at any time $\tau \geq \tau_0$, there is a set $Q(\tau)$ of t processes such that $p \notin Q(\tau)$ and a message broadcast by p at τ receives a response from each process of $Q(\tau)$ by time $\tau + \delta$ (where δ is a bound known by the processes). The very important point here is that the set $Q(\tau)$ of processes whose responses have to be received in a timely manner is not fixed and can be different at distinct times.

The notions of eventual t-source and eventual t-accessibility cannot be compared (which means that none of them can be simulated from the other). In a very interesting way these two notions have been combined in [12] where is defined the notion of *eventual t-moving source*. A process p is an eventual t-moving source if there is a time τ_0 such that at any time $\tau \geq \tau_0$ there is a set $Q(\tau)$ of t processes such that $p \notin Q(\tau)$ and a message broadcast

by p at τ is received by each process in $Q(\tau)$ by time $\tau+\delta$. As we can see, the *eventual t-moving source* assumption is weaker than the *eventual t-source* as the set $Q(\tau)$ can vary with τ.

Other time-based approaches are investigated in [7, 13]. They consider weak assumptions on both the initial knowledge of processes and the network behavior. Protocols building Ω are presented [7, 13] that assume the initial knowledge of each process is limited to its identity and the fact that no two identities are the same (so, a process knows neither n nor t). An unreliable broadcast primitive allows the processes to communicate. It is shown in [13] that Ω can be built as long as there is one correct process that can reach the rest of the correct processes via eventually timely paths; the corresponding protocol requires a correct process to send messages forever. Differently, one of the protocols presented in [7] is communication-efficient (after some time a single process has to send messages forever) while, as far as the network behavior is concerned, it only requires that each pair of correct processes be connected by fair lossy links, and there is a correct process whose output links to the rest of correct processes are eventually timely. This protocol is presented in Section 8.

The message pattern approach A totally different approach to build Ω has been introduced in [17]. That 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. The statement of such a property involves the system parameters n and t.

Let us assume that each process regularly broadcasts queries and, for each query, waits for the corresponding responses. Given a query, a response that belongs to the first $(n-t)$ responses to that query is said to be a *winning* response. Otherwise, the response is a *losing* response (then, that response is slow, lost or has never been sent because its sender has crashed). It is shown in [20] 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 processes such that $p \notin Q$ and 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 [17]. A protocol based on that time-free message pattern approach is presented in Section 7.

Combining both approaches This *message pattern* approach and the *eventual timely link* approaches cannot be compared. Interestingly, the message pattern approach and the eventual t-source approach have been combined in [21]. This combination shows that Ω can be implemented as soon as there is a correct process p such that there is a time τ_0 after which there is a set Q of t processes q such that $p \notin Q$ and either (1) each time a process $q \in Q$ broadcasts a query, it receives a winning response from p, or (2) the link from p to q is timely. As it can be seen, if only (1) is satisfied, we obtain the *message pattern* assumption, while, if only (2) is satisfied, we obtain the *eventual t-source* assumption. A even more general protocol based on weaker assumptions has recently been designed in [8].

More generally, here, the important fact is that the message pattern assumption and the timely link assumption are combined at the "finest possible" granularity level, namely, the link level, allowing thus the design of protocols with a better assumption coverage [22].

5 The *eventually synchronous* assumption

5.1 The additional assumption

The first additional assumption that we consider to enrich the base system model $AS_{n,t}[\emptyset]$ in order to implement an eventual leader service Ω is a relatively strong assumption as it considers that eventually the system has a synchronous behavior. More precisely, that assumption is the following [4, 6].

> "There are a finite time τ_0 and a bound δ, such that after τ_0, the transmission delay of any message is upper bounded by δ time units."

It is important to notice that neither τ_0 nor δ are a priori known, and they can never be explicitly known. The time τ_0 is called the *Global Stabilization Time*. It is important to notice that this assumption does not depend on the system parameter t. The corresponding system model is consequently denoted $AS_{n,n-1}[GST]$.

5.2 The LFA protocol

Several protocols that builds an eventual leader service in $AS_{n,n-1}[GST]$ have been proposed. We present here one of them that is particularly elegant. This algorithm, due to Larrea, Fernandez and Arevalo [15], is based on the following simple idea: elect the correct process that has the smallest pid. To that end, each process p_i considers only the processes with an identity j such $1 \leq j \leq i$ as candidates for being the eventual leader. The current leader of p_i (whose pid is kept in the local variable $leader_i$) is defined as the process p_j such that p_i suspects all the processes

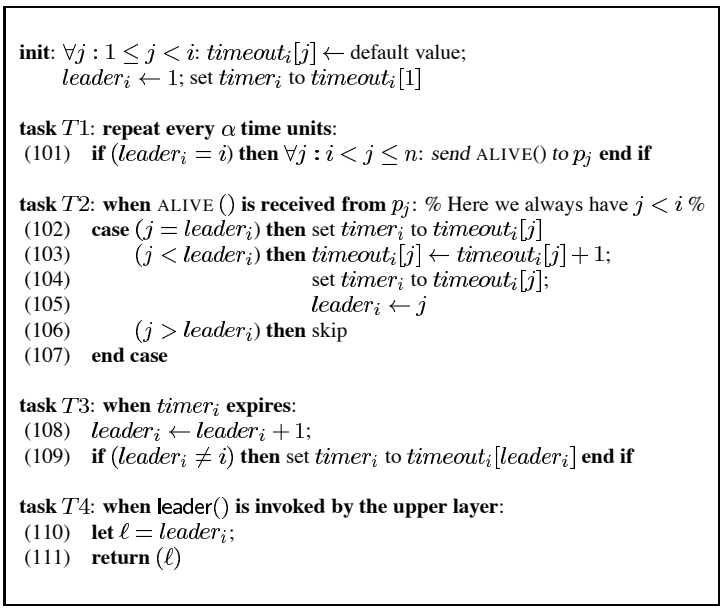

Figure 1. The LFA protocol [15] (code for p_i)

$p_1, p_2, \ldots, p_{j-1}$ to have crashed. It follows, that when a process p_i suspects all the processes $p_1, p_2, \ldots, p_{i-1}$, it considers it is the leader. When this occurs, p_i sends periodically an ALIVE() message to each process p_k with a higher identity (i.e., $k > i$) to inform them it is alive. A process p_i suspect another process p_j to have crashed by using a local timer (denoted $timer_i$) and a corresponding timeout value (kept in $timeout_i[j]$).

It is of course possible that, before the global stabilization time τ_0 occurs, a process p_i receives a message from a process p_x, such that $x < leader_i$. This means that p_i suspected (maybe erroneously) p_x to have crashed. In that case, in order to correct its mistake, p_i locally demotes its current leader p_{leader_i} and considers instead p_x as its new leader. The cause of such a mistake by p_i is due to the fact that p_i did not wait a long enough period before suspecting p_x. So, when this happens, in order to reduce the possibility of a future mistake with respect to p_x (i.e., $timer_i$ expires too early while p_x is p_i's current leader), p_i increases the corresponding timeout period $timeout_i[x]$.

The corresponding protocol is described in Figure 1. It is self-explanatory (α is a positive value that can be arbitrary).

5.3 Short discussion

A correctness proof of this protocol can be found in [15]: in all the runs of the system that satisfy the eventual synchrony assumption, there is a time after which all the alive processes have forever the same leader (that is an alive process). Actually, a more precise look into the way the protocol works shows a weaker synchrony and reliability assumption is sufficient, namely, it is sufficient that only the output links of the correct process with the smallest identity are eventually reliable and timely.

The protocol, that is not counter-based, requires a single type of message, and each message has to carry only the pid of its sender, which means that its size is $\log_2(n)$. It is easy to see that after an eventual leader p_ℓ has been elected, that process only sends messages forever. It follows that the protocol is *communication optimal* (as the process that is eventually elected as the single common leader has to forever indicate to the other processes that it is still alive).

6 The *eventual t-source* assumption

6.1 The additional assumption

This section considers a synchrony assumption weaker than the previous one [2].That assumption called "$\diamond t$-source" is the following.

> "There are a finite time τ_0, a correct process p, a bound δ, and t output channels of p, such that, after τ_0, any message sent by p on one of these channels is received at most δ units of time after it has been sent (such a channel is eventually δ-timely)."

Let us notice that it is not required that the destination processes associated with the t channels involved in the $\diamond t$-source be correct. Some -or all- of them can be faulty. This means that, as soon as t processes have crashed, any remaining process trivially becomes a t-source. Let $AS_{n,f}[\diamond t$-

source] denote a distributed system satisfying this synchrony assumption.

6.2 The ADFT protocol

The t-source assumption has been introduced by Aguilera, Delporte-Gallet, Fauconnier and Toueg in [2] where they also present an eventual leader election protocol for $AS_{n,f}[\diamond f\text{-source}]$. Its code for process p_i is described in Figure 2. Each process p_i manages an array $count_i[1..n]$. This array is such that $count_i[j]$ counts the number of suspicions of p_j as known by p_i. It is managed in such a way that it remains bounded when p_j is a correct $\diamond f$-source, while it increases without bound when p_j crashes. The leader is the process p_ℓ whose counter has the smallest value (task $T5$).

The key of the protocol is the management of each counter $count_i[j]$, i.e., the way such a counter is (or not) increased. To that end, each process p_i manages an array $suspect_i$ as follows (task $T4$): $suspect_i[j]$ keeps track of the set of processes that currently suspect p_j to have crashed (task $T3$). When this set contains $(n-t)$ processes, p_i considers there are enough processes that suspect p_j in order to increase $count_i[j]$. When this occurs p_i resets $suspect_i[j]$ to $\emptyset$.

As already indicated, the $\diamond t$-source assumption allows showing that every process p_j that crashes will be forever suspected (i.e., $count_i[j]$ will never stop increasing), while $count_i[j]$ remains bounded if p_j is a $\diamond$ t-source. Consequently, there is at least one entry of $count_i$ that remains bounded and all the entries of $count_i$ that remain bounded correspond to correct processes. The process that is elected by p_i is the process it suspects the less (as counted in $count_i[1..n]$).

Process identities are used to do a tie-break when there are several processes that are the less suspected. X being a set of pairs, the function $\min(X)$ returns the smallest pair of X. Let us recall that $(v,i) < (w,j)$ iff $(v < w) \vee (v = w \wedge i < j)$ (this is the classical lexicographical ordering).

6.3 Short discussion

A proof of the protocol can be found in [2]. It is easy to see an y correct process sends forever message. So, this protocol is not communication efficient. Moreover, some variables can increase forever.

7 The *eventual message pattern* assumption

This section presents a behavioral time-free assumption that allows implementing an eventual leader. This assumption is due to Mostefaoui, Mourgaya and Raynal [17].

7.1 The additional assumption

Query-response mechanism For our purpose, we consider that each process is provided with a query-response mechanism. Such a query-response mechanism can easily be implemented in a time-free distributed asynchronous system. More specifically, any process p_i can broadcast a QUERY_ALIVE() message and then wait for corresponding RESPONSE() messages from $(n-t)$ processes (these are the *winning* responses for that query). The other RESPONSE() messages associated with a query, if any, are systematically discarded (these are the *losing* responses for that query).

A query issued by p_i is *terminated* if p_i has received the $(n-t)$ corresponding responses it was waiting for. We assume that a process issues a new query only when the previous one has terminated. Without loss of generality, the response from a process to its own queries is assumed to always arrive among the first $(n-t)$ responses it is waiting for. Moreover, QUERY_ALIVE() and RESPONSE() are assumed to be appropriately tagged in order not to confuse RESPONSE() messages corresponding to different QUERY_ALIVE() messages.

The time-free assumption That assumption, denoted MP, is the following [17].

> "There are a time τ_0, a correct process p and a set Q of $(t+1)$ processes (τ_0, p and Q are not known in advance) such that, after τ_0, each process $p_j \in Q$ obtains a winning response from p to each of its queries (until p_j possibly crashes)."

As we can see, this assumption imposes no constraint on message transfer delays; those can increase forever. It is not based on a constraint related to physical time, but on a logical time notion, namely, a message delivery order. The corresponding system model is denoted $AS_{n,t}[MP]$.

7.2 The MMRT protocol

The following protocol (due to Mostefaoui, Mourgaya, Raynal and Travers [18]) is made up of three tasks executed by each process. Its underlying principles are relatively simple. It is based on the same heuristic as previously: each process elects as leader the process it suspects the less. To implement this idea (as in the protocol described in Figure 2) a process p_i manages an array $count_i[1..n]$ in such a way that $count_i[j]$ counts the number of times p_i suspects p_j to have crashed.

The aim of the tasks $T1$ and $T2$ is to manage the array $count_i$. To benefit from the MP property, the task $T1$ uses the underlying query-response mechanism. Periodically, each p_i issues a query and waits for the $(n-t)$ corresponding winning responses (lines 301-302). The response

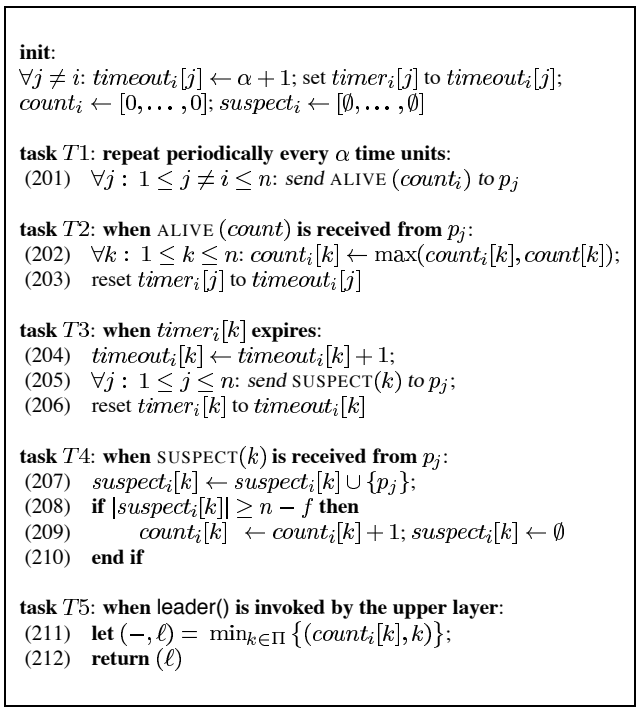

Figure 2. The ADFT protocol [2] (code for process p_i)

from p_j carries the set of processes that sent winning responses to its last query (this set is denoted rec_from_j). Then, according to the rec_from_j sets it has received, p_i updates accordingly its $count_i$ array.

The QUERY_ALIVE() messages implementing the query-response mechanism are used as a gossiping mechanism to disseminate the value of the $count_i$ array of each process p_i. This gossiping mechanism ensures that all the correct processes eventually elect the same leader.

7.3 Short discussion

A proof of this protocol can be found in [18]. It is important to observe that query-response "challenges" issued by different processes are independent one from the other. This has an interesting consequence, namely, a process can introduce an arbitrary delay before issuing a query-response challenge (line 301). Therefore, each process can, independently of the other processes, dynamically define and set such a delay to match the bandwidth that failure detector messages are allowed to use. As the protocol based on the t-source assumption (Figure 2), the protocol based on the MP assumption requires that each correct process sends messages forever (query-response mechanism).

8 Weak assumptions on initial knowledge, communication reliability and synchrony

All the previous protocols implicitly or explicitly assume that each process knows n and the identity of the other processes. So, one could ask whether these knowledge assumptions are necessary to implement an eventual leader service. This section shows that these implicit assumptions are not necessary. The ideas developed in the following have been introduced by Arevalo, Fernandez, Jimenez and Raynal in [7, 13].

8.1 A set of weak assumptions

The assumptions investigated in [7] are the following ones.

- Initial knowledge of processes. A process knows initially neither n, nor t, nor the identities of the other processes. It only knows its own identity, and the fact that the identities are totally ordered and no two processes have the same identity.
- Communication reliability. Each pair of correct processes is connected by a pair of typed fair lossy channels[2]. Moreover, there is a correct process whose out-

[2] A typed fair lossy channel is a channel such that, given an y message type, if it is used to transmit an infinite number of messages of that type,

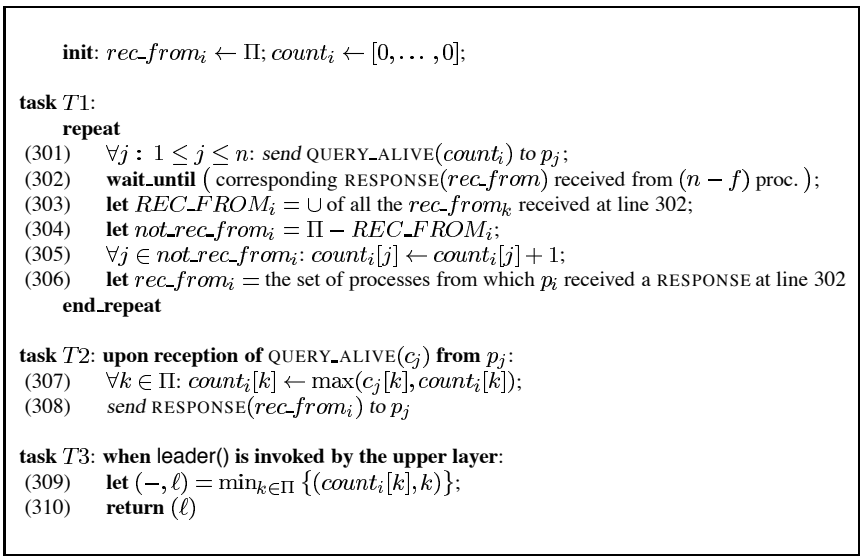

Figure 3. The MMRT protocol [18] (code for process p_i)

put channels to every correct process are eventually timely.

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

Let $AS_{n,n-1}[KCS]$ denote an asynchronous system whose runs satisfy the two previous assumptions.

8.2 The FJR protocol

A protocol that elects an eventual leader in $AS_{n,n-1}[KCS]$ is described in Figure 4. This protocol is due to Fernandez, Jimenez and Raynal [7]. The code of each process is made up of two tasks.

The task $T1$ That task is where, when it considers it is the leader, a process p_i sends messages in order not to be erroneously suspected by the other processes (line 401). 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 404). A message sent with a tag field equal to *heartbeat* (line 403) is called a heartbeat message; similarly, a message sent with a tag field equal to *stop_leader* (line 404) is called a stop_leader message.

Local variables Each process p_i maintains the following local variables.

- $members_i$: 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.
- $count_i[j]$ has the same meaning as in the previous protocols, namely, to p_i's knowledge, it represents the number of times that p_j has been suspected. It is managed in such a way that only p_i can increase $count_i[i]$; it does it when it receives a message from a process p_k indicating that p_k suspects it. Moreover, each message broadcast by p_i includes $count_i[i]$ in order each other process p_j updates $count_i[j]$.
- 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 $i \in$

it delivers an infinite number of these messages. This allows it to lose an infinite number of messages. As an example, let the infinite sequence of messages be the set of all the integers. If the channel lose all odd numbers and deliver all even numbers it is fair. From a practical point of view, this means that a simple retransmission mechanism allows transmitting any message in a reliable way.

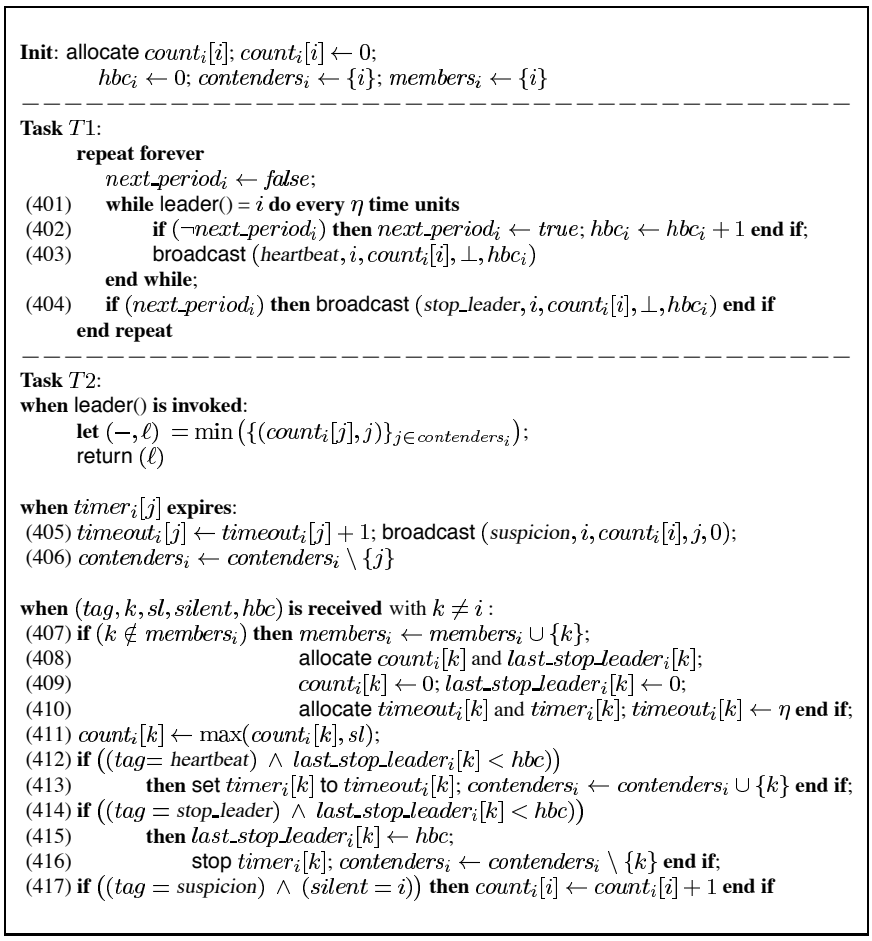

Figure 4. The FJR protocol (code for p_i)

$contenders_i$. This ensures that a leader election is not missed since p_i is always locally 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 401-404).

- 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 412) or a stop_leader message (line 414) 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 Each message $(tag, k, sl, silent, hbc)$ has five fields whose meaning is the following:

- The field tag 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 405.)

- The second field contains the id k of the message sender.

- sl is the value of $count_k[k]$ when p_k sent that message. Let us observe that the value of $count_k[k]$ can be disseminated only by p_k.

- $silent = j$ means that p_k suspects p_j to be faulty. Such a suspicion is due to a timer expiration that occurs at line 405. (Let us notice that the field $silent$ of a message that is not a suspicion message is always equal to $\perp$.)

- hbc: this field contains the value of the period counter hbc 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 $(suspicion, silent)$ defines a type.

Process behavior When a timer $timer_i[j]$ expires, p_i broadcasts a message indicating it suspects p_j (line 405), and accordingly suppresses j from $contenders_i$. Together with line 416, 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 407-410); p_i also updates $count_i[k]$ (line 411). Then, the processing of the message depends on its tag.

- The message is a heartbeat message (lines 412-413). 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 414-416). 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 417). If the suspicion concerns p_i, it increases accordingly $count_i[i]$.

8.3 Short discussion

The previous protocol is communication efficient (after some finite time a single process sends messages forever). Moreover, no messages carry values that increase without bound: be a run finite or infinite, the counters carried by the messages take a bounded number of values (the bound depends on the run).

9 Conclusion

The aim of this invited paper was to introduce the readers to issues one has to address when one has has to build reliable services on top of unreliable asynchronous systems. The paper focused on the eventual leader service. It has been shown that an asynchronous system has to satisfy additional synchrony assumptions in order non-trivial services can be built on top it. Several assumptions that allow implementing an eventual leader service have been visited and corresponding protocols presented. For more details on these assumptions or the protocols, the interested reader can consult the original papers that have been cited.

Acknowledgments

I would like to thank Ernesto Jimenez, Antonio Fernandez, Achour Mostefaoui, Eric Mourgaya and Corentin Travers for interesting discussions on the eventual leader election problem.

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

[7] 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 Computer Society Press, pp. 166-175, Philadelphia (PA), 2006.

[8] Fernández A. and Raynal M., From an intermittent rotating star to a leader. *Submitted to publication*, 2007.

[9] 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.

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

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

[12] Hutle M., Malkhi D., Schmid U. and Zhou L., Chasing the weakest system model for implementing Ω and consensus. *Brief Annoucement, Proc. 8th Int'l Symp. on Stabilization, Safety and Security in Distributed Systems (SSS'06)*, Springer-Verlag LNCS #4280, pp. 576-577, 2006.

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

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

[15] 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.

[16] 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.

[17] Mostéfaoui 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.

[18] 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.

[19] Mostéfaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2000.

[20] Mostéfaoui 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, 2004.

[21] 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.

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

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