Time-Free and Timer-Based Assumptions Can Be Combined to Obtain Eventual Leadership

Achour Mostefaoui, Michel Raynal, and Corentin Travers

Abstract—Leader-based protocols rest on a primitive able to provide the processes with the same unique leader. Such protocols are very common in distributed computing to solve synchronization or coordination problems. Unfortunately, providing such a primitive is far from being trivial in asynchronous distributed systems prone to process crashes. (It is even impossible in fault-prone purely asynchronous systems.) To circumvent this difficulty, several protocols have been proposed that build a leader facility on top of an asynchronous distributed system enriched with additional assumptions. The protocols proposed so far consider either additional assumptions based on synchrony or additional assumptions on the pattern of the messages that are exchanged. Considering systems with n processes and up to f process crashes, $1 \leq f < n$, this paper investigates the combination of a time-free assumption on the message pattern with a synchrony assumption on process speed and message delay. It shows that both types of assumptions can be combined to obtain a hybrid eventual leader protocol benefiting from the best of both worlds. This combined assumption considers a star communication structure involving $f + 1$ processes. Its noteworthy feature lies in the level of combination of both types of assumption that is "as fine as possible" in the sense that each of the f channels of the star has to satisfy a property independently of the property satisfied by each of the $f - 1$ other channels (the f channels do not have to satisfy the same assumption). More precisely, this combined assumption is the following: There is a correct process p (center of the star) and a set Q of f processes q ($p \notin Q$) such that, eventually, either 1) each time it broadcasts a query, q receives a response from p among the $(n - f)$ first responses to that query, or 2) the channel from p to q is timely. (The processes in the set Q can crash.) A surprisingly simple eventual leader protocol based on this fine grain hybrid assumption is proposed and proved correct. An improvement is also presented.

Index Terms—Asynchronous system, distributed algorithm, fault tolerance, hybrid protocol, leader election, process crash, time-free assumption, timer-based assumption.

1 Introduction

1.1 Context of the Study and Motivation

The design and implementation of reliable applications on top of asynchronous distributed systems prone to process crashes is a difficult and complex task. A main issue lies in the impossibility of correctly detecting crashes in the presence of asynchrony. In such a context, some problems become very difficult or even impossible to solve. The most famous of those problems is the *Consensus* problem for which there is no deterministic solution in asynchronous distributed systems where processes (even only one) may crash [9].

While consensus is considered as a "theoretical" problem, middleware designers are usually interested in the more practical *Atomic Broadcast* problem. That problem is both a communication problem and an agreement problem. Its communication part specifies that the processes can broadcast and deliver messages in such a way that each correct[1] process delivers at least the messages sent by the correct processes. Its agreement part specifies that there is a single delivery order (so, the correct processes deliver the same sequence of messages, and a faulty process delivers a prefix of this sequence of messages). It has been shown that *consensus* and *atomic broadcast* are equivalent problems in asynchronous systems prone to process crashes [4]: In such a setting, any protocol solving one of them can be used as a black box on top of which the other problem can be solved. Consequently, in asynchronous distributed systems prone to process crashes, the impossibility of solving consensus extends to atomic broadcast.

When faced with processing crashes in an asynchronous distributed system, the main problem comes from the fact that it is impossible to safely distinguish a crashed process from a process that is slow or with which communication is very slow. To overcome this major difficulty, Chandra and Toueg have introduced the concept of *Unreliable Failure Detector* [4]. Among the different classes of failure detectors, the class of leader oracles (denoted Ω and formally introduced by Chandra et al. in [5]) is at the core of several distributed agreement protocols. Such an oracle offers a leader() primitive that satisfies the following leadership property: A unique correct leader is eventually elected, but there is no knowledge on when this common leader is elected and, before this occurs, several distinct leaders (possibly conflicting) can coexist. Interestingly, it is possible to solve consensus (and related agreement problems) in asynchronous distributed systems equipped with such a "weak" oracle (as soon as these systems have a majority of correct processes) [5], [13], [17]. It has also been shown that Ω is the weakest failure detector class for solving consensus in these systems [5]. Unfortunately, Ω cannot be implemented in pure (time-free)

1. A *correct* process is a process that does not crash. See Section 2.

• *The authors are with IRISA, Université de Rennes 1, Campus de Beaulieu, Avenue du Général Leclerc, 35042 Rennes Cedex, France. E-mail: {mostefaoui, raynal, ctravers}@irisa.fr.*

Manuscript received 18 July 2005; revised 6 Jan. 2006; accepted 12 Jan. 2006; published online 25 May 2006.
Recommended for acceptance by D. Bader.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0337-0705.

asynchronous systems (its implementation would contradict the consensus impossibility result [9]).

Despite the impossibility to design leader protocols in pure time-free asynchronous systems, the design of a leader facility remains very attractive. More precisely, when we look from a "protocol design" side, the class of leader oracles has a noteworthy feature, namely, it allows the protocols that use such an oracle to benefit from a very nice property, namely, *indulgence* [10]. Let P be an oracle-based protocol and SP 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 SP. This means that each time P produces outputs, those are correct. (The periods during which outputs are provided are sometimes called *good* or *stable* periods [7].) Moreover, P always produces outputs when the underlying oracle meets its specification. The only case where P can be prevented from producing outputs is when the underlying oracle does not meet its specification (these periods are called *bad* or *unstable* periods).

Interestingly, Ω is a class of oracles that allows the design of indulgent consensus protocols [11]. The fact that the safety property SP of the Ω-based protocol P can never be violated, and the fact that its liveness property (outputs are produced) can be ensured in "good" periods, make attractive the design of indulgent Ω-based protocols and motivates the design of protocols that do "their best" to implement Ω within the asynchronous distributed system itself. A challenge is then to identify properties that, when satisfied by the asynchronous system, ensure that it evolves in a good period. This paper is on the design of protocols doing their best to build an eventual leader facility.

1.2 Related Work

Two main approaches have been investigated to implement leader oracles. The first, which we call "Timer-based," relies on the addition of timing assumptions [8]. Basically, this approach assumes that there are bounds on process speeds and message transfer delays, but these bounds are not known and hold only after some finite but unknown time. The protocols implementing an eventual leader facility in such "augmented" systems are based on timeouts (e.g., [1], [2], [14]). They use successive approximations to eventually provide each process with an upper bound on transfer delays and processing speed. They differ mainly in the "quantity" of additional synchrony they consider, and in the message cost they entail after a leader has been elected.

Among the protocols based on this approach, a protocol presented in [2] is particularly attractive, as it considers a very weak additional synchrony requirement. Let f be an upper bound on the number of processes that may crash ($1 \leq f < n$, where n is the total number of processes). This assumption is the following: The underlying asynchronous system, which can have fair lossy channels, is required to have a correct process p that is a $\diamond f\text{-}source$. This means that p has f output channels that are eventually timely: There is a time after which the transfer time of all the messages sent on such a channel is bounded (let us notice that this is trivially satisfied as soon as the receiver has crashed). Let us notice that such a $\diamond f\text{-}source$ is not known in advance and can never be explicitly known. It is also shown in [2] that there is no leader protocol if the system has only $\diamond(f-1)\text{-}sources$. (Other important issues such as "communication optimality" when implementing Ω are also investigated in [2].)

The second approach (introduced in [15] to implement the Chandra and Toueg's failure detectors defined in [4]) does not assume eventual bounds on process and communication delays. We call it "Message Pattern." It considers that there is a correct process p and a set Q of f processes (with $p \notin Q$, moreover, Q can contain crashed processes) such that, each time a process $q \in Q$ broadcasts a query, it receives a response from p among the first $(n-f)$ corresponding responses (such a response is called a winning response). It is easy to see that this assumption does not prevent message delays from always increasing without bound. Hence, it is incomparable with the synchrony-related $\diamond f\text{-}source$ assumption. This approach has been applied to the construction of a leader protocol in [18].

Let us observe that, when we address the problem from an abstract point of view, what seems intuitively needed to obtain an eventual leader protocol is the existence of a correct process p that is eventually no longer suspected by a set Q of f (correct or faulty) processes (with $p \notin Q$). When this occurs, there are $f+1$ processes (p + the processes in Q, forming a star centered at p) containing the information that p has not crashed. As a set of $f+1$ processes always includes at least one correct process, it becomes possible to envisage a protocol that is able to extract this information and make it visible to the whole set of processes. This tends to make us think that any protocol implementing Ω requires additional assumptions involving a set of $f+1$ processes.

1.3 Content of the Paper

When we look at the two previous approaches ($\diamond f\text{-}source$ and Message Pattern), we observe that they are orthogonal in the sense that one assumption cannot be used to simulate the other. So, an interesting question is the following: Is it possible to design an eventual leader protocol able to benefit from the best of both worlds, i.e., a protocol such that, as soon as one assumption is satisfied (regardless of which one, and whether the other is or is not satisfied), a leader is elected? Such a hybrid protocol would guarantee convergence if any one of the alternative assumptions is satisfied ($\diamond f\text{-}source$ or Message Pattern), thereby providing increased overall assumption coverage [20].

Such a combination of the two types of assumptions considers them separately in the sense that it considers that either the $\diamond f\text{-}source$ assumption is satisfied or the Message Pattern assumption is satisfied. So, a second and maybe more interesting question is the following: Is it possible to combine the two assumptions at a finer level, i.e., is it possible to have one assumption satisfied by a part of a system while another part of the system would satisfy the other assumption?

The paper answers this question by first introducing a hybrid assumption combining both types of assumption at a level "as fine as possible," and then presenting an eventual leader protocol based on this fine grain assumption. This combined assumption considers a star communication structure involving $f+1$ processes (these $f+1$ processes can differ from a run of the system to another run) and is such

that each of its f channels can satisfy a property independently of the property satisfied by the $f-1$ other channels. It is the following: There is a correct process p (center of the star) and a set Q of f processes q ($p \notin Q$) such that, eventually, either 1) each time it broadcasts a query, q receives a response from p among the $(n-f)$ first responses to that query, or 2) the channel from p to q is timely. (The processes in the set Q can crash.)

The fact that such a fine grain combination be possible is noteworthy for several reasons. First, it was not a priori evident that such a combination would be possible. It could have been the case that a fine combination of these two orthogonal types of assumption be incompatible. Second, at a conceptual level, it shows that there is some hidden unity in time-free and timer-based assumptions (as far as eventual leader election is concerned). More precisely, 1) a round trip corresponding to a query (from q to p) and the corresponding winning response (from p to q) used in the Message Pattern assumption type and 2) a timely message sent by p to q used in the Timer-based assumption type are both used to provide q with the same monitoring information on p. (A formal statement of such a unity remains a challenging open problem.) Finally, from a practical point of view, the channel-wise combination of the two types of assumption provides an assumption coverage better than any of these assumptions taken separately.

1.4 Organization of the Paper

The paper is made up of six sections. Section 2 presents the basic asynchronous computation model and the leader problem. Then, Section 3 presents the additional assumption (denoted $\mathcal{H}$) combining, in a simple and powerful way, a time-free assumption and a timer-based assumption. Section 4 presents a protocol based on $\mathcal{H}$, and Section 5 extends it to a more general context. Finally, Section 6 concludes the paper.

2 Basic Computation Model and Leadership Facility

2.1 Asynchronous Distributed System with Process Crash Failures

We consider a system consisting of a finite set Π of $n \geq 2$ processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process executes steps (a step is the reception of a set of messages with a local state change or the sending of messages with a local state change). It can fail by *crashing*, i.e., by prematurely halting. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash. A *faulty* process is a process that is not correct. As previously indicated, f denotes the maximum number of processes that can crash ($1 \leq f < n$).

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes (p_i, p_j) is connected by two directed channels, denoted $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$. 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, then, eventually, p_j receives that message unless it fails. There is no assumption about the relative speed of processes or message transfer delays (let us observe that channels are not required to be FIFO).

In the following, $AS_{n,f}[\emptyset]$ denotes an asynchronous distributed system made up of n processes among which up to $f < n$ can crash. More generally, $AS_{n,f}[P]$ denotes an asynchronous system made up of n processes among which up to $f < n$ can crash and satisfying the additional assumption P (so, $P = \emptyset$ means that the system is a *pure* asynchronous system).

We assume the existence of a global discrete clock. This clock is a fictional device which is not known by the processes; it is only used to state specifications or prove protocol properties. The range $\mathcal{T}$ of clock values is the set of natural numbers.

2.2 Leadership Facility

A *leader* oracle is a distributed entity that provides the processes with a function leader() that returns a process name each time it is invoked. A unique correct leader 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. The *leader* oracle (usually denoted Ω [5]) satisfies the following property:

- Eventual Leadership: There is a time t and a correct process p such that, after t, every invocation of leader() by any correct process returns p.

Ω-based consensus algorithms are described in [11], [13], [17][2] for systems where a majority of processes are correct ($f < n/2$). Such consensus algorithms can then be used as a subroutine to implement upper layer protocols such as atomic broadcast protocols (e.g., [4], [13], [16], [19]).

As consensus can be solved in an asynchronous system with a majority of correct processes and equipped with a leader oracle, and as consensus cannot be solved in purely asynchronous systems [9], it follows that a leader oracle cannot be implemented in an asynchronous system $AS_{n,f}[\emptyset]$ with $f < n/2$. Direct proofs of this impossibility that work for $f < n$ can be found in [2], [3], [18].[3] So, we have the following theorem:

Theorem 1 [2], [3], [18]. *$\forall\, f,\ 0 < f < n$, there is no protocol that implements a leader oracle in $AS_{n,f}[\emptyset]$.*

3 Considering Additional Assumptions

3.1 A Time-Free Additional Assumption: Notion of Winning Channel

3.1.1 Query-Response Mechanism

For our purpose (namely, the implementation of a leader facility), 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 $AS_{n,f}[\emptyset]$. More specifically, any process p_i can broadcast a QUERY() message and then wait for

2. The Paxos protocol [13] is leader-based and considers a more general model where processes can crash and recover, and links are fair lossy. (Its first version dates back to 1989, i.e., before the Ω formalism was introduced.)

3. These direct proofs do not rely on the impossibility to solve consensus in $AS_{n,f}[\emptyset]$ [9].

corresponding RESPONSE() messages from $(n-f)$ processes (these are the *winning* responses for that query, and the corresponding sender processes are the *winning* processes for that query). The other RESPONSE() messages associated with a query, if any, are systematically discarded (these are the *losing* responses for that query). The notion of winning/losing response is a time-free notion in the sense that its implementation does not require timers. (Of course, as the network bandwidth, physical time is an underlying resource needed to implement the query/response mechanism. But, these resources are not explicitly managed by the protocol.)

We assume that a process issues a new query only when it has received the $(n-f)$ winning responses corresponding to its previous query. Moreover, QUERY() and RESPONSE() messages are assumed to be implicitly tagged in order not to confuse RESPONSE() messages corresponding to different QUERY() messages.

Definition 1 (Eventually winning channel). *Let p_i and p_j be two processes. The directed channel $p_i \rightarrow p_j$ is* eventually winning *(denoted $\diamond WC$) if there is a time t such that the response from p_i to each query issued by p_j after t is a winning response (t is finite but unknown).*

Let us notice that, after p_j has crashed (if it ever crashes), it no longer issues queries. It follows that the channel $p_i \rightarrow p_j$ is then always winning, which means that a directed channel whose receiver is faulty is eventually winning.

Definition 2 (Assumption $x\text{-}\diamond WC$). *There is a correct process p_ℓ and a set Q of x processes p_j such that $p_\ell \notin Q$ and $\forall p_j \in Q$, the channel $p_\ell \rightarrow p_j$ is eventually winning.*

Let $AS_{n,f}[x\text{-}\diamond WC]$ denote an asynchronous system satisfying the property $x\text{-}\diamond WC$. Such a system can be seen as an asynchronous system in which there is a correct process (p_ℓ) that has x "favorite neighbors" (defining the set Q) that can communicate with it faster than with the other processes. When we consider the particular case $x = 1$, the $1\text{-}\diamond WC$ property boils down to a simple channel property, namely, there is a channel that is never the slowest among the channels connecting one of its endpoints to the other processes. Let us observe that, if x processes crash, $AS_{n,f}[\emptyset]$ trivially satisfies the $x\text{-}\diamond WC$ assumption.

It is shown in [15] that, eventually, strong failure detectors (denoted $\diamond\mathcal{S}$) can be implemented in $AS_{n,f}[x\text{-}\diamond WC]$ when $x \geq f$. ($\diamond\mathcal{S}$ and Ω have the same computational power as far as process crash failures are concerned [5], [6].)

3.2 A Timer-Based Additional Assumption: Notion of Timely Channel

Definition 3 (Eventually timely channel). *Let p_i and p_j be two processes. The directed channel $p_i \rightarrow p_j$ is* eventually timely *(denoted $\diamond TL$) if there is a time t after which 1) p_j has crashed or 2) there is a bound δ such that each message sent by p_i after t is received by p_j within δ units of time (t and δ are not known.)*

The intent of this definition is that a recipient never receives a *late* message. This is trivially the case for a crashed recipient, hence the first item in the previous definition.

Let us observe that, in order to be able to try determining whether a directed channel $p_i \rightarrow p_j$ is eventually timely, the basic underlying asynchronous system has to be enriched with additional assumptions and mechanisms to allow realizing meaningful time measurements. So, to address the eventual timer-based property of a channel $p_i \rightarrow p_j$, we assume that 1) p_i and p_j have local clocks (that can accurately measure time intervals[4]) and 2) there is a lower and upper bound on the execution rate (number of steps per time unit) of p_i and p_j.

Definition 4 (Assumption $x\text{-}\diamond TL$).[5] *There is a correct process p_ℓ and a set Q of x processes p_j such that $p_\ell \notin Q$ and $\forall p_j \in Q$, the channel $p_\ell \rightarrow p_j$ is eventually timely.*

As before, let $AS_{n,f}[x\text{-}\diamond TL]$ denote an asynchronous system with synchronous processes[6] and satisfying the assumption $x\text{-}\diamond TL$ and where each process is equipped with a local clock with which it can accurately measure time intervals (local clocks are not required to be synchronized). Such a system has a correct process (p_ℓ) that has x "favorite neighbors" (defining the set Q), "favorite" in the sense that eventually these processes never timeout when waiting for its messages. (Using the terminology of [2], p_ℓ is an eventual x-source.) It is shown in [2] that the leader facility Ω can be implemented in $AS_{n,f}[x\text{-}\diamond TL]$ when $x \geq f$. (It is important to notice that, to benefit from the $f\text{-}\diamond TL$ assumption, the Ω protocol described in [2] explicitly uses the parameters n and f in its code.)

As discussed before, let us notice that, when we consider the time free assumptions $x\text{-}\diamond WC$, $0 \leq x \leq n-1$ (respectively, the timer based assumptions $x\text{-}\diamond TL$, $0 \leq x \leq n-1$), $f \leq x$ is a necessary requirement to implement a leader facility Ω in $AS_{n,f}[x\text{-}\diamond WC]$ (respectively, in $AS_{n,f}[x\text{-}\diamond TL]$).

3.3 Combining Time-Free and Timer-Based Assumptions

Let us observe that a system $AS_{n,f}[f\text{-}\diamond WC]$ does not prevent message delays from always increasing, from which we conclude that $AS_{n,f}[f\text{-}\diamond TL]$ cannot be simulated from $AS_{n,f}[f\text{-}\diamond WC]$. Similarly, $AS_{n,f}[f\text{-}\diamond WC]$ cannot be obtained from $AS_{n,f}[f\text{-}\diamond TL]$, as it is possible that the response from a correct process be always a losing response. This means that, in the general case, the assumptions $f\text{-}\diamond WC$ and $f\text{-}\diamond TL$ are not equivalent.

To benefit from the best of both worlds, an interesting approach (investigated in [18]) consists in designing a leader protocol that works in a system that satisfies any of these assumptions $f\text{-}\diamond WC$ or $f\text{-}\diamond TL$, i.e., in a system

4. The clock of p_i is used to periodically send messages. The clock of p_j is used to set timers. If several processes have local clocks, it is not required that these clocks be synchronized. They only need to be accurate in measuring time intervals.

5. Our $x\text{-}\diamond TL$ notion is the same as the $\diamond x\text{-}source$ notion that, as noticed in Section 1, has been first proposed in [2], where a more general communication model is considered (namely, a weaker model with fair lossy channels).

6. This means that there is a lower and upper bound on the execution rate (number of steps per time unit) of each nonfaulty process. This synchrony assumption is necessary for the processes be able to exploit the fact that some channels are timely [2].

$AS_{n,f}[f\text{-}\diamond WC \vee f\text{-}\diamond TL]$. Surprisingly, as announced in Section 1, it is possible to do much better: These assumptions can actually be combined in a finer way to build a leadership facility. This combination is defined as an assumption denoted $f\text{-}(\diamond WC \vee \diamond TL)$ (in short $\mathcal{H}$).

Definition 5 (Assumption $\mathcal{H}$). *There is a correct process p_ℓ and a set Q of f processes p_x such that $p_\ell \notin Q$ and $\forall p_x \in Q$, the channel $p_\ell \rightarrow p_x$ is eventually timely or eventually winning.*

An asynchronous system that satisfies $\mathcal{H}$ will be denoted $AS_{n,f}[\mathcal{H}]$ in the rest of the paper. As we can see, the assumption $\mathcal{H}$ combines the time-free and timer-based assumptions on a channel base: Some channels connecting a correct process p_c can be timely while others can be winning. This is very interesting as the "granularity" of the combined assumption is "weaker" than each assumption taken separately:[7] The output channels $p_c \rightarrow p_x$ of a base correct process p_c do not have to be simultaneously either timely or winning; each of them can satisfy any assumption. The next section shows how $\mathcal{H}$ can be exploited to design a leader protocol. It is interesting to notice that $\mathcal{H}$ is trivially satisfied in the runs where f processes crash.

3.3.1 Local Property versus Nonlocal Property

It is interesting to notice that the eventually timely channel property is a *local* property in the sense that it characterizes the behavior of a channel independently of the behavior of the other channels. Expressed differently, the eventually winning channel property is not a local property as the fact that a channel is eventually winning involves other channels (the eventually winning channel "wins" with respect to other channels). The statement of this *nonlocal* property involves n and f, which are global parameters appearing in the definition of the system model, namely, $AS_{n,f}[\emptyset]$.

3.3.2 The Case $f = 1$

For the systems where at most one process may crash (an interesting case, in practice), the assumption $\mathcal{H}$ becomes: There is a pair of processes p and q such that, after some unknown but finite time, 1) the channel $p \rightarrow q$ becomes timely or 2) for each of its query, q receives the response from p among the $(n-1)$ first responses corresponding to that query.

This assumption can be rephrased as follows: There is a pair of processes p and q such that, eventually, 1) the channel $p \rightarrow q$ is timely or 2) for each query issued by q, the round-trip delay of the corresponding QUERY/RESPONSE messages exchanged with p is never the largest among the n round-trip delays of all the QUERY/RESPONSE messages associated with that query. We have shown in [15] that the probability that part 2) will be satisfied in practice is very close to 1. This means that, when $f = 1$, the probability that $\mathcal{H}$ will be satisfied is practically equal to 1.

7. From a more operational perspective, we can say that an eventually timely channel satisfies a *push* property, while an eventually winning channel satisfies a *pop* property. Here, "pop" means that a process receives a response message because it has first sent a query, while "push" means that a process receives messages without before asking for them.

This suggests choosing a pair (p, q) of processes (any pair can be chosen) and connecting this pair with two channels $p \rightarrow q$ and $q \rightarrow p$ whose communication speed is higher than the communication speed of the other channels connecting p or q to the other processes. This creates an asymmetry within the system that allows implementing Ω despite asynchrony and the crash of $f = 1$ process. It follows that consensus can be solved despite one crash in an asynchronous distributed system equipped with such a pair of channels.

4 A Basic $\mathcal{H}$-Based Asynchronous Leader Protocol

4.1 Underlying Principles

The $\mathcal{H}$-based protocol described in Fig. 1 is surprisingly simple. It relies on the following principles: The aim is for each process p_i to manage an array $count_i[1:n]$ such that $count_i[j]$ will remain bounded if p_j is correct and p_i trusts it (not to have crashed). Then (Task $T3$), given such an array, p_i considers as the current leader the process p_ℓ such that $count_i[\ell]$ has the smallest value (when two counters have the smallest value, process ids are used to break ties).

To get a $count_i$ array with consistent values, each process p_i uses two distinct sets of data structures, one addressing the "timely" part of the assumption $\mathcal{H}$, the other one addressing its "winning response" part. More precisely, we have the following:

- On the "timely" side, the Boolean array $timely_i[1:n]$ is the relevant data structure. It is managed as follows:

  First, each process p_j sends periodically ALIVE() messages (Task $T2$) to inform the others that it has not crashed. Accordingly, each process p_i manages a timeout value ($timeout_i[j]$) and a timer ($timer_i[j]$) with the hope that it will receive the next ALIVE() message from p_j before that timeout value has elapsed. If a timeout occurs, p_i sets $timely_i[j]$ to $false$ (lines 17-18). When it receives an ALIVE() message from p_j, p_i resets the timer and increases the timeout value if the channel from p_j was considered as not being timely (with respect to the previous timeout value). After p_i has received a message from p_j, we always have $timely_i[j] = true$.

  So, the idea of the previous mechanism is to obtain the following property: The processes p_j such that eventually $timely_i[j]$ remains permanently equal to $true$ determine a set of timely channels $p_j \rightarrow p_i$.
- On the "query-response" side, the relevant data structure is the Boolean array $winning_i[1:n]$.

  Repeatedly (Task $T1$), each process p_i sends a QUERY() message to all the other processes and waits for the first $(n-f)$ RESPONSE(). The processes that sent these winning responses are defined as the "winning processes" for that query (line 5).

  So, the query-response mechanism is used to get the following property: If a process p_j is such that, eventually, $winning_i[j]$ remains permanently equal to $true$, then p_j will no longer be suspected by p_i.

```
init: trusted_i ← {i}; count_i ← [0, ..., 0]; timely_i[i] ← true;
      for_each j ≠ i do timely_i[j] ← false; timeout_i[j] ← α + 1; set timer_i[j] to timeout_i[j] end_do

task T1: % query/response task for determining winning channels %
(1) repeat for_each j do winning_i[j] ← false; send QUERY(count_i) to p_j end_do;
(2)       wait until ( corresponding RESPONSE(trusted) received from ≥ (n − f) processes );
(3)       let TRUSTED_i = ∪ the sets trusted_j received at line 2;
(4)       for_each j ∈ Π − TRUSTED_i do count_i[j] ← count_i[j] + 1 end_do;
(5)       for_each j such_that (∃ RESPONSE() from p_j at line 2) do winning_i[j] ← true end_do;
(6)       trusted_i ← { k | winning_i[k] ∨ timely_i[k] }
(7) end repeat

task T2: % repeating task for determining timely channels %
(8) repeat every α time units:
(9)       for_each j ≠ i do send ALIVE() to p_j end_do

task T3: % server task %
(10) upon reception of QUERY(count_j) from p_j :
(11)      for_each k ∈ Π do count_i[k] ← max(count_i[k], count_j[k]) end_do;
(12)      send RESPONSE(trusted_i) to p_j

(13) upon reception of ALIVE() from p_j :
(14)      if (¬timely_i[j]) then timeout_i[j]← timeout_i[j] + 1 end_if;
(15)      timely_i[j] ← true;
(16)      reset timer_i[j] to timeout_i[j]

(17) upon expiration of timer_i[j] :
(18)      timely_i[j] ← false

(19) upon a leader() invocation by the upper layer:
(20)      let ℓ such that (count_i[ℓ], ℓ) = min_{k∈Π} {(count_i[k], k)};
(21)      return (ℓ)
```

Fig. 1. Leader protocol module associated with p_i.

A key element of the protocol is the way the Boolean arrays $timely_i$ and $winning_i$ are combined and used by p_i to update $count_i[1:n]$. Their combination is done at line 6: p_i trusts the processes p_j such that $timely_i[j] \vee winning_i[j]$. The aim of the set $trusted_i$ is to include the processes that eventually have timely channels towards p_i or whose responses to p_i's queries are eventually always winning.

When a process p_j sends a response to p_i (line 12), it uses the current value of its set $trusted_j$ to inform p_i that it (p_j) does not currently suspect these processes. When it has received the $(n-f)$ winning responses it was waiting for, p_i trusts all the processes trusted by the winning processes (line 3) and suspects the other processes by increasing their counter accordingly (line 4).

Finally, in order for all the correct processes to have the same bounded entries in their $count_i$ arrays (and, consequently, be able to elect the same leader), each QUERY() message piggybacks the $count_j$ array of its sender p_j (line 1), thereby allowing the receiver p_i to update its array $count_i$ (line 11).

Remark. 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 (at line 1). 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.

4.2 Proof of the $\mathcal{H}$-Based Protocol

With C denoting the set of processes that are correct in a given run that satisfies $\mathcal{H}$, let us consider the following set definitions (PL stands for "Potential Leaders"):

$$PL = \{p_x \mid \exists p_i \in C : \ count_i[x] \text{ is bounded}\},$$
$$\forall\, p_i \in C : \ \ PL_i = \{p_x \mid count_i[x] \text{ is bounded}\}.$$

The proof is made up of three parts:

- We first show that, in any run that satisfies $\mathcal{H}$, the set PL is not empty (Lemma 1).
- We then show that PL is a subset of C, the processes that are correct in that run (Lemma 2).
- Finally, we show that $PL_i = PL$ for any correct process p_i (Lemma 3).

Lemma 1. $\mathcal{H} \Rightarrow (PL \neq \emptyset)$.

Proof. Let p_ℓ be a correct process that satisfies the hybrid assumption $\mathcal{H}$. This means that there is a time t and a set Q of f processes p_x (not including p_ℓ) such that, after t, 1) the channel $p_\ell \rightarrow p_x$ is timely or 2) p_x receives only winning responses from p_ℓ to its queries. Remember that Q can contain crashed processes, as both the constraints 1) or 2) are trivially satisfied for a crashed process p_x. As a crashed process can trivially be a member of Q, we consider, in the following, that no process crashes to make easier the formulation of the proof (without having to consider particular cases).

Let Q_T be the subset of processes p_x of Q, such that, after t, the channel $p_\ell \rightarrow p_x$ is timely. Due to the definition of "eventual timely channel" and to lines 13-18, there is a time $t1_x \geq t$ after which $timely_x[\ell]$ remains continuously $true$. Let $t1 \geq \max_{p_x \in Q_T}(t1_x)$.

Similarly, let Q_W be the subset of processes p_x of Q, such that, after t, p_x receives only winning responses from p_ℓ to its queries. Due to the query-response mechanism and to line 5, there is a time $t2_x \geq t$ after which $winning_x[\ell]$ remains continuously $true$. Let $t2 \geq \max_{p_x \in Q_W}(t2_x)$. Finally, let $t3 = \max(t1, t2)$.

After $t3$, there is a set of f processes p_x such that (for the ones of them that have not crashed) $p_\ell \in trusted_x$. Moreover, due to the protocol code, $timely_\ell[\ell]$ is always equal to $true$ (it is initialized to that value and never modified thereafter, as no process sets a timer with respect to itself); consequently, we always have $p_\ell \in trusted_\ell$. So, after $t3$, considering the f processes of Q plus p_ℓ, we have $f+1$ processes p_y such that $p_\ell \in trusted_y$.

Let $t4 \geq t3$, a time after which all the faulty processes have crashed. As no process blocks forever at line 2, each correct process p_i executes the lines 1-6 infinitely often and, after $t4$, it always receives at least one RESPONSE ($trusted$) message with $p_\ell \in trusted$ (as it waits for $(n-f)$ RESPONSE() messages and $f+1$ processes p_y are such that $p_\ell \in trusted_y$). Consequently, after $t4$ and for any correct process p_i, we have $p_\ell \in TRUSTED_i$ at line 3, from which we conclude that no correct process p_i increments $count_i[\ell]$ at line 4. Let M_ℓ be the greatest value among the values of the $count_i[\ell]$ variables of the correct processes p_i at time $t4$.

Finally, as after $t4$, there are only correct processes; the only way for $count_i[\ell]$ to be increased is at line 11. Due to the gossiping of the $count_i$ arrays (line 1 and line 11), it follows that the variable $count_i[\ell]$ of each correct process p_i becomes equal to M_ℓ and then keeps that value forever. □

The next corollary follows directly from the gossiping of the $count_i$ arrays:

Corollary 1. *Let p_i and p_j be any pair of correct processes. If, after some time, $count_i[k]$ remains forever equal to some constant value M_k, then there is a time after which $count_j[k]$ remains forever equal to the same value M_k.*

Lemma 2. $PL \subseteq C$.

Proof. We show the contrapositive, i.e., if p_j is a faulty process, then each correct process p_i is such that $count_i[j]$ increases forever.

Let $t0$ be a time after which all the faulty process have crashed. As a crashed process does not send ALIVE() messages, it follows, from the management of the $timely_i[j]$ Boolean variables at lines 13-18, that there is a time $t1 \geq t0$ after which, $\forall p_i \in C$ and $\forall p_j \in \Pi - C$, we continuously have $timely_i[j] = false$. Similarly, as a crashed process does not send RESPONSE() messages, it follows, from the lines 1-6, that there is a time $t2 \geq t0$ after which, $\forall p_i \in C$ and $\forall p_j \in \Pi - C$ we continuously have $winning_i[j] = false$.

This means that there is a time $t3 \geq \max(t1, t2)$ after which, $\forall p_i \in C$ and $\forall p_j \in \Pi - C$, we have forever $p_j \notin trusted_i$ (line 6). As, after $t3$, no RESPONSE() message carries a set $trusted_i$ containing p_j, it follows that there is a time $t \geq t3$ after which no faulty process p_j belongs to the set $TRUSTED_i$ of a correct process p_i. Consequently, each time after t at which a correct process p_i executes line 4, it increases $count_i[j]$. As there are at most f faulty processes, no correct process can block forever at line 2. Consequently, a correct process executes line 2 infinitely often, and $count_i[j]$ never stops increasing. □

Lemma 3. $p_i \in C \Rightarrow PL_i = PL$.

Proof. Let us first observe that $PL = \bigcup_{p_i \in C} PL_i$ (this follows immediately from the definition of PL). Consequently, $PL_i \subseteq PL$. The inclusion in the other direction is an immediate consequence of Corollary 1. □

Theorem 2. *$\forall f > n$, the protocol described in Fig. 1 implements a leader facility in $AS_{n,f}[\mathcal{H}]$.*

Proof. The proof follows directly from Lemmas 1, 2, and 3, which state that all the correct processes have the same nonempty set of potential leaders, which includes only correct processes. Moreover, due to Corollary 1, all the correct processes have the same counter values for the processes of PL (and those values are the only ones to be bounded). It follows that the correct processes elect the same leader that is the correct process with the smallest counter value. □

4.3 Discussion

As already noticed, $\mathcal{H}$ is always satisfied in the runs where f processes do crash (as any of the assumptions $f\text{-}\diamond WC$ and $f\text{-}\diamond TL$ is then satisfied).

Let us say "p_x is a process that makes satisfied the assertion $\mathcal{H}$" when p_x is a correct process such that, after some time, there is a set of f processes that receive from p_x only timely messages or winning responses. It is important to notice that the process that is eventually elected is not necessarily a process p_x that makes satisfied the assertion $\mathcal{H}$. Moreover, the star communication structures that make $\mathcal{H}$ satisfied in different runs can be distinct.

The reader can check that the protocol works in more runs than the ones caught by the assumption $\mathcal{H}$. Let p_k be a process that, after some time, is trusted forever (i.e., $count_j[k]$ is bounded at any correct process p_j), and let p_i be one of the f processes such that the directed channel $p_k \rightarrow p_i$ is either eventually timely or eventually winning. As shown at line 6, it is sufficient for the protocol to work that this channel be always trusted by p_i (i.e., $k \in trusted_i$): it can be timely during some periods and winning during other periods. The important point is that the Boolean formula $timely_i[k] \vee winning_i[k]$ be equal to $true$ each time it is evaluated at line 6 (it is not required that the channel $p_k \rightarrow p_i$ eventually be always "timely" or always "winning," it is only required to eventually be always "timely OR winning").[8] Providing a clean and formal statement of such an assumption $\mathcal{H}'$ (that is weaker than $\mathcal{H}$) remains a challenging problem.

8. In more "engineering" terms, the quality of service offered by such a channel is *adaptive* as it is not required to be—after some time—always timely or always winning, it can only be forever alternating between timely (detected with a push mechanism) and winning (detected with a pop mechanism).

init: $count_i \leftarrow [0, \ldots, 0]$; $timely_i[i] \leftarrow true$;
for_each (d, j) **such that** $0 \leq d \leq f, j \in \Pi$ **do** $trusted_i[d, j] \leftarrow false$ **end_do**; $trusted_i[0, i] \leftarrow true$;
$trusted\_winning_i[i][0, i] \leftarrow true$; $trusted\_timely_i[i][0, i] \leftarrow true$;
for_each $j \neq i$ **do** $timely_i[j] \leftarrow false$; $timeout_i[j] \leftarrow \alpha + 1$; set $timer_i[j]$ to $timeout_i[j]$ **end_do**

task $T1$: % query/response task for determining winning channels %
(1) **repeat for_each** j **do** $winning_i[j] \leftarrow false$; *send* QUERY$(count_i)$ *to* p_j **end_do**;
(2) **wait until** (corresponding RESPONSE$(trusted)$ received from $\geq (n - f)$ processes);
(3)′ **let** $TRUSTED_i = \{j \mid \exists\, 0 \leq d \leq f : trusted_k[d, j]$ where $trusted_k$ has been received at line 2 $\}$;
(4) **for_each** $j \in \Pi - TRUSTED_i$ **do** $count_i[j] \leftarrow count_i[j] + 1$ **end_do**;
(5)′ **for_each** j **such_that** ($\exists$ RESPONSE() from p_j at line 2) **do** $winning_i[j] \leftarrow true$;
$trusted\_winning_i[j] \leftarrow trusted_j$ **end_do**;
(6)' **for_each** (d, j) **such that** $0 < d \leq f, j \in \Pi$ **do**
$trusted_i[d, j] \leftarrow (\exists\, k \mid (winning_i[k] \wedge trusted\_winning_i[k][d-1, j])$
$\vee\ (timely_i[k] \wedge trusted\_timely_i[k][d-1, j]))$ **end_do**
(7) **end repeat**

task $T2$: % repeating task for determining timely channels %
(8) **repeat every** α **time units**:
(9)′ **for_each** $j \neq i$ **do** *send* ALIVE$(trusted_i)$ *to* p_j **end_do**

task $T3$: % server task %
(10) **upon reception of** QUERY$(count_j)$ **from** p_j :
(11) **for_each** $k \in \Pi$ **do** $count_i[k] \leftarrow \max(count_i[k], count_j[k])$ **end_do**;
(12) *send* RESPONSE$(trusted_i)$ *to* p_j

(13)′ **upon reception of** ALIVE$(trusted_j)$ **from** p_j :
(14) **if** $(\neg timely_i[j])$ **then** $timeout_i[j] \leftarrow timeout_i[j] + 1$ **end_if**;
(15)′ $timely_i[j] \leftarrow true$; $trusted\_timely_i[j] \leftarrow trusted_j$;
(16) reset $timer_i[j]$ to $timeout_i[j]$

(17) **upon expiration of** $timer_i[j]$:
(18) $timely_i[j] \leftarrow false$

(19) **upon a** leader() **invocation by the upper layer**:
(20) **let** ℓ **such that** $(count_i[\ell], \ell) = \min_{k \in \Pi} \{(count_i[k], k)\}$;
(21) **return** (ℓ)

Fig. 2. $\mathcal{H}^+$-based module associated with p_i.

5 Toward a More General Assumption

5.1 From a Star to a "Tree"

It is, of course, possible for a system to have runs that do not satisfy $\mathcal{H}$, i.e., runs without a communication star made up of $f + 1$ processes, centered at a correct process p_ℓ, and such that, for each of its channels $p_\ell \rightarrow p_x$, eventually, either 1) the channel from p_ℓ to p_x is timely or 2) each time p_x broadcasts a query, it receives a winning response from p_ℓ.

Despite the fact that $\mathcal{H}$ cannot be satisfied, it is still possible to implement Ω if the runs of the system satisfy a weakened version of $\mathcal{H}$ defined as follows. The star is now a virtual star made up of f virtual channels $p_\ell \rightarrow p_x$ such that, for each virtual channel, the abstract property that the virtual channel $p_\ell \rightarrow p_x$ is supposed to satisfy (by being eventually timely or eventually winning at the operational level), is actually satisfied by a directed path connecting p_ℓ to p_x (e.g., the path $p_\ell \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow p_x$). This observation brings us to the following definition of a weakened version of $\mathcal{H}$, that we call $\mathcal{H}^+$.[9] (Instead of a star, $\mathcal{H}^+$ considers a tree whose inner nodes are correct processes.)

9. A similar weakening was proposed first in [2], where only the "eventually timely channel" assumption is considered. No corresponding protocol is explicitly given. A protocol based on a flooding technique is only suggested.

Definition 6 (Assumption $\mathcal{H}^+$). *There is a correct process p_ℓ and a set Q of f processes p_x, such that $p_\ell \notin Q$ and $\forall p_x \in Q$, there is directed path of processes $p_\ell = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_{y-1} \rightarrow q_y = p_x$ such that the intermediary processes $q_1, q_2, \ldots, q_{y-1}$ are correct and each directed channel $q_z \rightarrow q_{z+1}, 0 \leq z > y$, is eventually timely or eventually winning.*[10]

$\mathcal{H}^+$ allows the design of a protocol implementing an eventual leader facility. (If we denote $\mathcal{H}^d$ the instance of the assumption $\mathcal{H}^+$ in which the maximal length of a directed path is limited to d, we have $\mathcal{H}^1 = \mathcal{H}$.)

5.2 An Improved Protocol

The $\mathcal{H}^+$-based protocol (Fig. 2) is the protocol of Fig. 1 with a simple modification that concerns the set $trusted_i$ and the way it is managed. In addition to a new line in the initialization part, the lines that are modified are marked with ′ in Fig. 2.

- $trusted_i$ is now a Boolean array whose dimensions are $[0 : f, 1 : n]$; so, it is made up of $n \times (f + 1)$ bits. A row is a distance d, while a column is a process identity j. The only distances that are considered are the distances from 0 to f, as, in the worst case, the

10. Let us notice that it follows from this definition that the processes $q_1, \ldots, q_{y-1}$ involved in a directed path can belong to Q.

$f+1$ processes involved in the $\mathcal{H}^+$ assumption define a single path connecting p_ℓ (the unknown distinguished process) to p_i.

The meaning of this array is the following: $trusted_i[d,j] = true$ means that, from p_i point of view, there is a directed path of length d connecting p_j to p_i, and this path is made up of noncrashed processes and timely or winning channels.

This Boolean matrix represents the knowledge that p_i has on the "good" paths connecting processes to itself. Initially, only $trusted_i[0,i]$ is set to $true$, as p_i always "trusts" itself.

- When a process p_j sends an ALIVE () message, that message now carries the current value of the array $trusted_j$ of its sender p_j (line 9′).

  Accordingly, when p_i receives such a message (line 13′), it keeps the Boolean array in an additional local variable denoted $trusted\_timely_i[j]$ (line 15′). In that way, p_i not only considers the channel $p_j \rightarrow p_i$ as timely, but also the paths that p_j considers as "good" paths.

  The same is done when, after it has issued a query, p_i receives a winning response from p_j. It stores the paths that p_j considers as "good" paths in $trusted\_winning_i[j]$ (line 5′).

  These additional data structures and their simple management allow p_i to know which the processes and the paths trusted by each process p_j that it trusts are (because the channel $p_j \rightarrow p_i$ is currently perceived as timely or winning).
- After it has received all the winning responses associated with a query, a process p_i now defines $TRUSTED_i$ as the set of the processes p_j that are trusted by the processes p_k that sent these winning responses (line 3′). The processes trusted by such a process p_k are the processes p_j such that $trusted_k[d,j]$.
- Finally, the core of the modification appears at line 6′ where a process p_i redefines the set of processes it trusts, i.e., where the Boolean array $trusted_i[0:f,1:n]$ is recomputed. $trusted_i[d,j]$ is set to the value $true$ if and only if there is a trusted "immediate neighbor" p_k (i.e., a process p_k such that $timely_i[k]$ or $winning_i[k]$ is true) that trusts p_j at distance $d-1$.

Remark. Let us notice that the protocol described in Fig. 2 can easily be adapted to systems whose communication is provided by a sparse network. These systems are such that each process p_i is directly connected to only a set of neighbors nb_i. The underlying communication graph has to be strongly k-connected with $k > f$, and the minimum in-degree has to be $\geq k$ [12].

5.3 Proof of the $\mathcal{H}^+$-Based Protocol

The proof that the protocol is correct is close to the previous one. The proofs of Lemma 2, Corollary 1, and Theorem 2 are still valid. The new proof of Lemma 2 is very close to the previous one: It only has to take into account the new way the set of trusted processes is computed. Lemma 1 has to be reformulated to consider the premises $\mathcal{H}^+$ instead of $\mathcal{H}$. Its new proof follows:

Lemma 4. $\mathcal{H}^+ \Rightarrow (PL \neq \emptyset)$.

Proof. Let p_ℓ be the correct process and Q be the set of processes that make $\mathcal{H}^+$ satisfied (see Definition 6). Let $Q' = Q \cup \{p_\ell\}$. Similarly to Lemma 1, as a crashed process can trivially be a member of Q', to make the formulation of the proof easier (without having to consider particular cases), we consider in the following that no process crashes.

We claim (*Claim C1*) that, after some finite time, each process p_x in Q' is such that $\exists\, d:\ trusted_x[d,\ell] = true$. The rest of the proof is similar to the one of Lemma 1. Indeed, from the claim *C1*, we know that there is a time t after which $f+1$ processes p_x are such that $\exists\, d:\ trusted_x[d,\ell] = true$. As no process blocks forever at line 2, each correct process p_i executes the lines $1-6'$ infinitely often, and, after t, it always receives at least one RESPONSE ($trusted$) message such that $\exists\, d : trusted\ [d,\ell] = true$. Consequently, after t and for any correct process p_i, we have $p_\ell \in TRUSTED_i$ at line 3′, from which we conclude that no correct process p_i increments $count_i[\ell]$ at line 4.

Let M_ℓ be the greatest value among the values of the $count_i[\ell]$ variables of the correct processes p_i at time t. Finally, as after t there are only correct processes, the only way for $count_i[\ell]$ to be increased is at line 11. Due to the gossiping of the $count_i$ arrays (line 1 and line 11), it follows that the variable $count_i[\ell]$ of each correct process p_i becomes equal to M_ℓ and then keeps that value forever. The theorem follows.

Claim C1: After some finite time, each process p_x in Q' is such that $\exists\, d:\ trusted_x[d,\ell] = true$.

Proof of Claim C1. Let us first notice that, for any process p_y, we initially have $trusted_y[0,y] = true$. Moreover, $trusted_y[0,y]$ is never updated at line 6′. Taking $y = \ell$, we conclude that $trusted_\ell[0,\ell] = true$ is always satisfied.

Let us now consider $p_x \in Q'$ with $p_x \neq p_\ell$. As 1) there is a finite path of correct processes from p_ℓ to p_x, e.g., $p_\ell = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_k = p_x$, 2) the length of this path is at most d, 3) each directed channel of this path $q_z \rightarrow q_{z+1}$, $0 \leq z > k$, is eventually timely or eventually winning, and 4) the fact that, after each channel of the finite path has become timely or winning, the Boolean value $trusted_\ell[0,\ell]$—that is always equal to $true$—is forwarded from process to process from p_ℓ to p_x via this sequence of channels, we conclude from line 6′ that there is a time after which $\exists d$ such that $trusted_x[d,\ell]$ remains permanently equal to $true$. *End of the proof of Claim C1*.□

6 Conclusion

Leader-based protocols are common in distributed computing. They rely on an underlying primitive that provides the same unique leader to the processes. Such a primitive is usually used to solve synchronization or coordination problems. While it is particularly easy to implement a leader primitive in a fault-free system, its construction in an asynchronous system prone to process crashes is impossible if the underlying system is not enriched with additional assumptions. While the traditional approach to build a distributed leader facility in such crash-prone asynchronous

systems considers a single type of additional assumption, namely, either a synchrony assumption (timer-based assumption) or an assumption on the message exchange pattern (time-free assumption), this paper has shown that a hybrid approach benefiting from the best of both kinds of assumption is possible, meaningful, and attractive.

More precisely, considering systems with n processes and up to f process crashes, $1 \leq f < n$, this paper has proposed a combination of a time-free assumption on the message pattern with a synchrony assumption on process speed and message delay. It has presented a very general hybrid protocol benefiting from the best of both worlds. This combined assumption considers a star communication structure involving $f+1$ processes. Its noteworthy feature lies in the level of combination of both types of assumption that is "as fine as possible" in the sense that each of the f channels of the star has to satisfy a property independently of the property satisfied by each of the $f-1$ other channels (the f channels do not have to satisfy the same assumption). More precisely, this combined assumption is the following: There is a correct process p (center of the star) and a set Q of f processes q (distinct from p) such that, eventually, either 1) each time it broadcasts a query, q receives a response from p among the $(n-f)$ first responses to that query, or 2) the channel from p to q is timely. (The processes in the set Q can crash.)

Interestingly, the protocol based on the time-free/timely hybrid assumption is not only particularly simple, but, due to the fact that it combines several assumption types, it also provides an assumption coverage better than the one offered by any protocol based on a single of these assumptions.

Acknowledgments

The authors are grateful to the referees for their useful comments that helped them improve the presentation of the paper. They would also like to thank Marcos Aguilera for interesting discussions on the the notion of local property versus nonlocal property, and David Powell for suggesting the word *winning response*.

References

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

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

[3] E. Anceaume, A. Fernandez, A. Mostefaoui, G. Neiger, and M. Raynal, "Necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors," *J. Computer and System Sciences*, vol. 68, pp. 123-133, 2004.

[4] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 2, pp. 225-267, 1996.

[5] T.D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *J. ACM*, vol. 43, no. 4, pp. 685-722, 1996.

[6] F. Chu, "Reducing Ω to $\diamond\mathcal{W}$," *Information Processing Letters*, vol. 76, no. 6, pp. 293-298, 1998.

[7] F. Cristian and C. Fetzer, "The Timed Asynchronous System Model," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 6, pp. 642-657, June 1999.

[8] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, pp. 288-323, 1988.

[9] M.J. Fischer, N. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374-382, 1985.

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

[11] R. Guerraoui and M. Raynal, "The Information Structure of Indulgent Consensus," *IEEE Trans. Computers*, vol. 53, no. 4, pp. 453-466, Apr. 2004.

[12] M. Hutle, "Omega in Sparse Networks," *Fast Abstracts, Proc. 10th IEEE Int'l Pacific Rim Dependable Computing Symp. (PRDC '04)*, 2004.

[13] L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems*, vol. 16, no. 2, pp. 133-169, 1998.

[14] M. Larrea, A. Fernández, and S. Arévalo, "Optimal Implementation of the Weakest Failure Detector for Solving Consensus," *Proc. 19th Symp. Reliable Distributed Systems (SRDS '00)*, pp. 52-60, 2000.

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

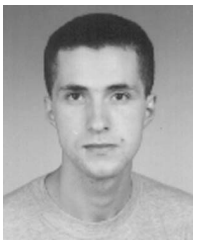
[16] A. Mostefaoui and M. Raynal, "Low-Cost Consensus-Based Atomic Broadcast," *Proc. Seventh IEEE Pacific Rim Int'l Symp. Dependable Computing (PRDC '00)*, pp. 45-52, 2000.

[17] A. Mostefaoui and M. Raynal, "Leader-Based Consensus," *Parallel Processing Letters*, vol. 11, no. 1, pp. 95-107, 2001.

[18] A. Mostefaoui, M. Raynal, and C. Travers, "Crash-Resilient Time-Free Eventual Leadership," *Proc. 23rd Symp. Reliable Distributed Systems (SRDS '04)*, pp. 208-217, 2004.

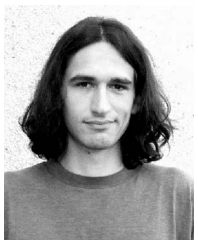
[19] F. Pedone and A. Schiper, "Handling Message Semantics with Generic Broadcast Protocols," *Distributed Computing*, vol. 15, no. 2, pp. 97-107, 2002.

[20] D. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing (FTCS-22)*, pp. 386-395, 1992.



Achour Mostefaoui received the engineer degree in computer science in 1990 from the University of Algiers and the PhD degree in computer science in 1994 from the University of Rennes, France. He is currently an associate professor in the Computer Science Department at the University of Rennes, France. His research interests include fault-tolerant distributed systems, group communication, consistency in DSM systems, and distributed checkpointing. He has published more than 65 scientific publications and served as a reviewer for more than 30 journals and international conferences. He is heading the Software Engineering Department of the University of Rennes.

Michel Raynal has been a professor of computer science since 1981. At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he founded a research group on distributed algorithms in 1983. His research interests include distributed algorithms, distributed computing systems, networks, and dependability. His main interest lies in the fundamental principles that underlie the design and the construction of distributed computing systems. He has been the principal investigator of a number of research grants in these areas and has been invited all over the world to give lectures on distributed computing systems. Professor Raynal has published more than 90 papers in journals (*Journal of the ACM*, *Acta Informatica*, *Distributed Computing*, *Communications of the ACM*, *Information and Computation*, *Journal of Computer and System Sciences*, *Journal of Parallel and Distributed Computing*, *IEEE Transactions on Computers*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Computer*, *IEEE Software*, IPL, PPL, *Theoretical Computer Science*, *Real-Time Systems Journal*, *The Computer Journal*, etc.), and more than 180 papers in conferences (ACM STOC, ACM PODC, ACM SPAA, IEEE ICDCS, IEEE DSN, DISC, IEEE IPDPS, Europar, FST&TCS, IEEE SRDS, SIROCCO, LATIN, etc.). He has also written seven books devoted to parallelism, distributed algorithms, and systems (MIT Press and Wiley). He has served on program committees for more than 90 international conferences (including ACM PODC, DISC, ICDCS, DSN, SRDS, etc.) and chaired the program committee of more than 15 international conferences. From 2002-2004, he chaired the steering committee leading the DISC symposium series. He received the IEEE ICDCS Best Paper Award three times in a row: 1999, 2000, and 2001. Recently, he cochaired the the SIROCCO 2005 Conference (devoted to communication complexity) and the IWDC 2005 conference. Michel Raynal is a general cochair of IEEE ICDCS 2006.



Corentin Travers is currently a PhD student at IRISA, a CNRS-INRIA-University joint research center in Rennes, France, advised by Professor Michel Raynal. His research interests include fault-tolerant distributed computing, distributed algorithms, and the theory of asynchronous distributed computing. He has published papers in the IEEE SRDS, LATIN, and OPODIS international conferences.