Robust Stabilizing Leader Election*

Carole Delporte-Gallet[1], Stéphane Devismes[2], and Hugues Fauconnier[1]

[1] LIAFA, Université D. Diderot (France)
{cd,hf}@liafa.jussieu.fr
[2] LaRIA, Université de Picardie Jules Verne (France)
stephane.devismes@u-picardie.fr

Abstract. We mix two approaches of the fault-tolerance: *robustness* and *stabilization*. Using these approaches, we propose leader election algorithms that tolerate both *transient* and *crash failures*. Our goal is to show the implementability of the robust self- and/or pseudo- stabilizing leader election in various systems with weak reliability and synchrony assumptions. We try to propose, when it is possible, *communication-efficient* implementations. Also, we exhibit some assumptions required to obtain robust stabilizing leader election algorithms. Our results show that the gap between robustness and stabilizing robustness is not really significant when we consider fix-point problems such as leader election.

1 Introduction

Two kinds of faults are usually considered: the *transient* and *crash* failures. The *stabilization* introduced by Dijkstra in 1974 [2] is a general technique to design algorithms tolerating transient failures. However, such stabilizing algorithms are usually not *robust*: they do not withstand crash failures. Conversely, *robust* algorithms are usually not designed to go through transient failures (*n.b.*, some robust algorithms tolerate the loss of messages, *e.g.*, [3]). There is some papers that deal with both stabilization and crash failures, *e.g.*, [4,5,6,7]. In [4], Gopal and Perry propose techniques for transforming robust protocols in a synchronous network into robust self-stabilizing versions. Beauquier and Kekkonen-Moneta introduce in [6] the first self-stabilizing failure detector implementation in a synchronous system. In [5], authors prove that robust self-stabilization cannot be achieved in asynchronous networks for a wide range of problems including leader election even when self-stabilization or robustness alone can be done.

We are interested in designing leader election algorithms that tolerate transient and crash failures. Actually, we focus on finding stabilizing solutions in message-passing with the possibility of some process crashes. The impossibility results in [8,5] constraints us to make some assumptions on the link and process synchrony. So, we look for the weakest assumptions allowing to obtain stabilizing leader election algorithm in a system where some processes may crash.

Leader election has been extensively studied in both stabilizing (*e.g.*, [9,10]) and robust (*e.g.*, [11,12]) areas. In particular, note that in the robust systems,

* The full version of this paper in available on the HAL server, see [1].

T. Masuzawa and S. Tixeuil (Eds.): SSS 2007, LNCS 4838, pp. 219–233, 2007.
© Springer-Verlag Berlin Heidelberg 2007

leader election is also considered as a failure detector. Such a failure detector, called Ω, is important because it has been shown in [13] that it is the weakest failure detector with which one can solve the consensus.

The notion of stabilization appears with the concept of *self-stabilization*: a *self-stabilizing algorithm*, regardless of the initial configuration of the system, guarantees that the system reaches in a finite time a configuration from which it *cannot* deviate from its intended behavior. In [14], Burns *et al* introduced the more general notion of *pseudo-stabilization*. A pseudo-stabilizing algorithm, regardless of the initial configuration of the system, guarantees that the system reaches in a finite time a configuration from which it *does not* deviate from its intended behavior. These two notions guarantee the *convergence* to a correct behavior. However, the self-stabilization also guarantees that since the system recovers a *legitimate* configuration, it remains in a *legitimate* configuration forever: the *closure* property. In contrast, a pseudo-stabilizing algorithm only guarantees an *ultimate closure*: the system can move from a *legitimate* configuration to an *illegitimate* one but eventually it remains in a *legitimate* configuration forever.

We study the problem of implementing robust self- and/or pseudo- stabilizing leader election in various systems with weak reliability and synchrony assumptions. We try to propose, when it is possible, *communication-efficient* implementations: an algorithm is *communication-efficient* if it eventually only uses $n-1$ unidirectionnal links (where n is the number of processes), which is optimal [15]. Communication-efficiency is quite challenging in the stabilizing area because stabilizing implementations often require the use of heartbeats which are heavy in terms of communication. In this paper, we first show that the notions of immediate synchrony and eventually synchrony are in some sense equivalent concerning the stabilization. Hence, we only consider synchrony properties that are immediate. In the systems we study: (1) all the processes are synchronous and can communicate with each other but some of them may crash and, (2) some links may have some synchrony or reliability properties. Our starting point is a full synchronous system noted $\mathcal{S}_5$. We show that a self-stabilizing leader election can be communication-efficiently done in such a system. We then show that such strong synchrony assumptions are required in the systems we consider to obtain a self-stabilizing communication-efficient leader election. Nevertheless, we also show that a self-stabilizing leader election that is not communication-efficient can be obtained in some weaker systems: any system where there exists at least one path of synchronous links between each pair of alive processes ($\mathcal{S}_3$) and any system having a *timely bi-source*[1] ($\mathcal{S}_4$). In addition, we show that we cannot implement any self-stabilizing leader election without these assumptions. Hence, we then consider the pseudo-stabilization. We show that communication-efficient pseudo-stabilizing leader election can be done in various weak models: any system $\mathcal{S}_4$ and any system having a *timely source*[2] and *fair* links ($\mathcal{S}_2$). Using a previous result of Aguilera *et al* ([3]), we recall that communication-efficiency

[1] A timely bi-source is a synchronous process having all its links that are synchronous.

[2] A timely source is a synchronous process having all its output links that are synchronous.

Table 1. Implementability of the robust stabilizing leader election

	$\mathcal{S}_5$	$\mathcal{S}_4$	$\mathcal{S}_3$	$\mathcal{S}_2$	$\mathcal{S}_1$	$\mathcal{S}_0$
Communication-Efficient Self-Stabilization	Yes	No	No	No	No	No
Self-Stabilization	Yes	Yes	Yes	No	No	No
Communication-Efficient Pseudo-Stabilization	Yes	Yes	?	Yes	No	No
Pseudo-Stabilization	Yes	Yes	Yes	Yes	Yes	No

cannot be done if we consider now systems having at least *one timely source* but where the *fairness* of all the links is not required ($\mathcal{S}_1$). However, we show that a non-communication-efficient pseudo-stabilizing solution can be implemented in such systems. Finally, we conclude with the basic system where all links can be asynchronous and lossy ($\mathcal{S}_0$): the leader election can be neither pseudo- nor self-stabilized in such a system ([8,5]). Table 1 summarizes our results.

It is important to note that the solutions we propose are essentially adapted from previous existing robust algorithms provided, in particular, in [11,3]. Actually, the motivation of the paper is not to propose new algorithms. Our goal is merely to show some required assumptions to obtain self- or pseudo- stabilizing leader election algorithms in systems where some processes may crash. In particular, we focus on the borderline assumptions where we go from the possibility to have self-stabilization to the possibility to have pseudo-stabilization only. Another interesting aspect of adaptating previous existing robust algorithms is to show that, for fix-point problems[3] such as leader election, the gap between robustness and stabilizing robustness is not really significant: in such problems, adding the stabilizing property is quite easy.

Roadmap. In the next section, we present the model for our systems. We then consider the problem of the robust stabilizing leader election in various kinds of systems (Sections 3 to 8). We conclude with the future works in Section 9.

2 Preliminaries

2.1 Distributed Systems

We consider *distributed systems* where each process can communicate with each other through *directed links*: there is a directed link from each process to all the others. We denote the *communication network* by the digraph $G = (V, E)$ where $V = \{1,...,n\}$ is the set of n processes ($n > 1$) and E the set of directed links. A collection of distributed *algorithms* run on the system. These algorithms can be seen as automata that enable processes to coordinate their activities. We modelize the *executions* of a distributed *algorithm* $\mathcal{A}$ in the system $\mathcal{S}$ by the pair $(\mathcal{C},\mapsto)$ where $\mathcal{C}$ is the set of configurations and $\mapsto$ is a collection of binary transition relations on $\mathcal{C}$ such that for each transition $\gamma_{i-1} \mapsto \gamma_i$ we have $\gamma_{i-1} \neq \gamma_i$. A configuration consists in the state of each process and the collection of messages

[3] Roughly speaking, a problem is a fix-point problem if the problem can be expressed by some invariant properties of some variables.

in transit at a given time. The state of a process is defined by the values of its variables. An *execution* of $\mathcal{A}$ is a *maximal* sequence $\gamma_0,\tau_0,\gamma_1,\tau_1,\ldots,\gamma_{i-1},\tau_{i-1},\gamma_i,\ldots$ such that $\forall i \geq 1$, $\gamma_{i-1} \mapsto \gamma_i$ and the transition $\gamma_{i-1} \mapsto \gamma_i$ occurs after time elapse τ_{i-1} time units ($\tau_{i-1} \in \mathbb{R}^+$). For each configuration γ in any execution e, we denote by $\overrightarrow{e_\gamma}$ the suffix of e starting in γ, $\overleftarrow{e_\gamma}$ denotes the associated prefix. We call *specification* a particular set of executions.

2.2 Self- and Pseudo- Stabilization

Definition 1 (Self-Stabilization [2]). *An algorithm $\mathcal{A}$ is* self-stabilizing *for a specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if in any execution of $\mathcal{A}$ in $\mathcal{S}$, there exists a configuration γ such that any suffix starting from γ is in $\mathcal{F}$.*

Definition 2 (Pseudo-Stabilization [14]). *An algorithm $\mathcal{A}$ is* pseudo-stabilizing *for a specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if in any execution of $\mathcal{A}$ in $\mathcal{S}$, there exists a suffix that is in $\mathcal{F}$.*

Robust Stabilization. Here, we not only consider the transient failures: our systems may go through transient and crash failures. We assume that some processes may be crashed in the initial configuration. We also assume that the links are not necessary reliable during the execution. In the following, we will show that despite these constraints, it is possible (under some assumptions) to design stabilizing algorithms. Note that the fact that we only consider initial crashes is not a restriction (but rather an assumption to simplify the proofs) because we focus on the leader election which is a fix-point problem: in such problems, the safety properties do not concern the whole execution but only a suffix.

2.3 Informal Model

Processes. Processes execute by taking steps. In a step a process executes two actions in sequence: (1) either it tries to receive one message from another process, or sends a message to another process, or does nothing, and then (2) changes its state. A step need not to be instantaneous, but we assume that each action of a step takes effect at some instantaneous moment during the step. The configuration of the system changes each time some steps take effect: if there is some steps that take effect at time t_i, then the system moves from a configuration γ_{i-1} to another configuration γ_i ($\gamma_{i-1} \mapsto \gamma_i$) where γ_{i-1} was the configuration of the system during some time interval $[t_{i-1}, t_i[$ and γ_i is the configuration obtained by applying on γ_{i-1} all actions of the steps that take effect at time t_i.

A process can fail by permanently crashing, in which case it definitively stops to take steps. A process is *alive at time t* if it is not crashed at time t. Here, we consider that all processes that are alive in the initial configuration are alive forever. An alive process executes infinitely many steps. We consider that any subset of processes may be crashed in the initial configuration.

We assume that the execution rate of any process cannot increase indefinitively: there exists a non-null lower bound on the time required by the alive

processes to execute a step[4]. Also, every alive process is assumed to be *timely*: it satisfies a non-null upper bound on the time it requires to execute each step. Finally, our algorithms are structured as a *repeat forever* loop with a bounded number of steps in each loop iteration. So, each alive process satisfies a lower and an upper bound, resp. noted α and β, on the time it needs to execute an iteration of its *repeat forever* loop. We assume that each process knows α and β.

Links. Processes can send messages over a set of directed links. There is a directed link from each process to all the others. A message m carries a *type* T in addition to its *data* D: $m = (T,D) \in \{0,1\}^* \times \{0,1\}^*$. For each incoming link (q,p) and each type T, the process p has a message buffer, $\texttt{Buffer}_p[q,T]$, that can hold at most one *single* message of type T. $\texttt{Buffer}_p[q,T] = \perp$ when it holds no message. If q sends a message m to p and the link (q,p) does not lose m, then $\texttt{Buffer}_p[q,T]$ is eventually set to m. When it happens, we say that *message m is delivered to p from q* (*n.b.*, we make no assumption on the delivrance order). If $\texttt{Buffer}_p[q,T]$ was set to some previous message, this message is then overwritten. When p takes a step, it may choose a process q and a type T to read the contents of $\texttt{Buffer}_p[q,T]$. If $\texttt{Buffer}_p[q,T]$ contains a message m (*i.e.*, $\texttt{Buffer}_p[q,T] \neq \perp$), then we say that *$p$ receives m from q* and $\texttt{Buffer}_p[q,T]$ is reset to $\perp$.

A link (p,q) is *timely* if there exists a constant δ such that, for every execution and every time t, each message m sent to q by p at time t is delivered to q from p within time $t + \delta$ (any message that is initially in a timely link is delivered within time δ). A link (p,q) is *eventually timely* if there exists a constant δ for which every execution satisfies: there is a time t such that every message m that p sends to q at time $t' \geq t$ is delivered to q from p by time $t' + \delta$ (any message that is already in an eventually timely link at time t is delivered within time $t + \delta$). We assume that every process knows δ. We also assume that $\delta > \beta$. A link which is neither timely nor eventually timely can be arbitrary slow, or can lose messages. A *fair* link (p,q) satisfies: for each type of message T, if p sends infinitely many messages of type T to q, then infinitely many messages of type T are delivered to q from p. A link (p,q) is *reliable* if every message sent by p to q is eventually delivered to q from p.

Particular Caracteristics. A *timely source* (resp. an *eventually timely source*) [3] is an alive process having all its *output links* that are *timely* (resp. *eventually timely*). A *timely bi-source* (resp. an *eventually timely bi-source*) [16] is an alive process having all its (input and output) *links* that are *timely* (resp. *eventually timely*). We call *timely routing overlay* (resp. *eventually timely routing overlay*) any strongly connected graph $G' = (V',E')$ where V' is the subset of all alive processes and E' a subset of *timely* (resp. *eventually timely*) links.

Finally, note that the notions of *timeliness* and *eventually timeliness* are "equivalent" in (pseudo- or self-) stabilization in a sense that every stabilizing algorithm in a system $\mathcal{S}$ having some timely links is also stabilizing in the system $\mathcal{S}'$ where $\mathcal{S}'$ is the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$, and reciprocally (see Theorems 1 and 2).

[4] Except for the first step that we allow to not satisfy this lower bound.

Theorem 1. *Let $\mathcal{S}$ be a system having some timely links. Let $\mathcal{S}'$ be the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$. An algorithm $\mathcal{A}$ is pseudo-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if $\mathcal{A}$ is pseudo-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}'$.*

Proof. By definition, a timely link is also an eventually timely link. Hence, we trivially have: if $\mathcal{A}$ is pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}'$, then $\mathcal{A}$ is also pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$.

Assume now that $\mathcal{A}$ is pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$ but not pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}'$. Then, there exists an execution e of $\mathcal{A}$ in $\mathcal{S}'$ such that no suffix of e is in $\mathcal{F}$. Let γ be the configuration of e from which all the eventually timely links of $\mathcal{S}'$ are timely. As no suffix of e is in $\mathcal{F}$, no suffix of $\overrightarrow{e_\gamma}$ is in $\mathcal{F}$ too. Now, $\overrightarrow{e_\gamma}$ is a possible execution of $\mathcal{A}$ in $\mathcal{S}$ because (1) γ is a possible initial configuration of $\mathcal{S}$ ($\mathcal{S}$ and $\mathcal{S}'$ have the same set of configurations and any configuration can be initial) and (2) every eventually timely link of $\mathcal{S}'$ is timely in $\overrightarrow{e_\gamma}$. Hence, as no suffix of $\overrightarrow{e_\gamma}$ is in $\mathcal{F}$, $\mathcal{A}$ is not pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$ — a contradiction. □

Following a proof similar to the one of Theorem 1, we have:

Theorem 2. *Let $\mathcal{S}$ be a system having some timely links. Let $\mathcal{S}'$ be the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$. An algorithm $\mathcal{A}$ is self-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if $\mathcal{A}$ is self-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}'$.*

Communication-Efficiency. An algorithm is *communication-efficient* [11] if there is a time from which it uses only $n-1$ unidirectional links.

Systems. We consider six systems denoted by $\mathcal{S}_i$, $i \in [0...5]$ (see Figure 1). All these systems satisfy: (1) the value of the variables of every alive process can be arbitrary in the initial configuration, (2) every link can initially contain a finite but unbounded number of messages, and (3) except if we explicitly state, each link between two alive processes is neither fair nor timely (we just assume that the messages cannot be corrupted). The system $\mathcal{S}_0$ corresponds to the basic system where no further assumptions are made: in $\mathcal{S}_0$, the links can be arbitrary slow or lossy. In $\mathcal{S}_1$, we assume that there exists at least one timely source (whose identity is unknown). In $\mathcal{S}_2$, we assume that there exists at least one timely source (whose identity is unknown) and every link is fair. In $\mathcal{S}_3$, we assume that there exists a timely routing overlay. In $\mathcal{S}_4$, we assume that there exists at least one timely bi-source (whose identity is unknown). In $\mathcal{S}_5$, all links are timely.

2.4 Robust Stabilizing Leader Election

In the leader election, each process p has a variable $Leader_p$ that holds the identity of a process. Intuitively, eventually all alive processes should hold the identity of the same process forever and this process should be alive. Formally, there exists an alive process l and a time t such that at any time $t' \geq t$, every alive process p satisfies $Leader_p = l$.

System	Properties
$\mathcal{S}_0$	*Links*: arbitrary slow, lossy, and initially not necessary empty *Processes*: can be initially crashed, timely forever otherwise *Variables*: initially arbitrary assigned
$\mathcal{S}_1$	$\mathcal{S}_0$ with at least one *timely source*
$\mathcal{S}_2$	$\mathcal{S}_0$ with at least one *timely source* and every link is *fair*
$\mathcal{S}_3$	$\mathcal{S}_0$ with a *timely routing overlay*
$\mathcal{S}_4$	$\mathcal{S}_0$ with at least one *timely bi-source*
$\mathcal{S}_5$	$\mathcal{S}_0$ except that all links are *timely*

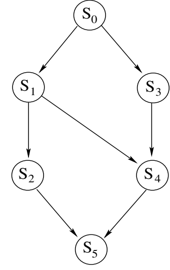

Fig. 1. Systems considered in this paper ($\mathcal{S} \to \mathcal{S}'$ means $\mathcal{S}' \subset \mathcal{S}$)

3 Communication-Efficient Self-Stabilizing Leader Election in $\mathcal{S}_5$

We first seek a communication-efficient self-stabilizing leader election algorithm in system $\mathcal{S}_5$. To get the communication-efficiency, we proceed as follows: Each process p periodically sends ALIVE to all other processes *only if it thinks to be the leader*, *i.e.*, *only if* $Leader_p = p$ (Lines 16-18 of Algorithm 1).

Any process p such that $Leader_p \neq p$ always chooses as leader the process from which it receives ALIVE the most recently (Lines 6-13). When a process p such that $Leader_p = p$ receives ALIVE from q, it sets $Leader_p$ to q if $q < p$ (Lines 6-13). By this method, there eventually exists at most one alive process p such that $Leader_p = p$.

Finally, every process p such that $Leader_p \neq p$ uses a *counter* that is incremented at each loop iteration to detect if there is no alive process q such that $Leader_q = q$ (Lines 21-27). When the counter becomes greater than a well-chosen value, p can deduce that there is no alive process q such that $Leader_q = q$. In this case, p simply elects itself by setting $Leader_p$ to p (Line 24) in order to guarantee the liveness of the election: in order to ensure that there eventually exists at least one process q such that $Leader_q = q$.

To apply the previously described method, Algorithm 1 uses only one message type: ALIVE and two *counters*: $SendTimer_p$ and $ReceiveTimer_p$. Any process p such that $Leader_p = p$ uses the counter $SendTimer_p$ to periodically send ALIVE to the other processes. $ReceiveTimer_p$ is used by each process p to detect when there is no alive process q such that $Leader_q = q$. These counters are incremented at each iteration of the *repeat forever* loop in order to evaluate a particular time elapse. Using the lower and upper bound on the time to execute an iteration of this loop (*i.e.*, α and β), each process p knows how many iterations it must execute before a given time elapse passed. For instance, a process p must count $\lceil \delta/\alpha \rceil$ loop iterations to wait at least δ times.

Theorem 3. *Algorithm 1 implements a communication-efficient self-stabilizing leader election in system* $\mathcal{S}_5$*.*

Algorithm 1. Communication-Efficient Self-Stabilizing Leader Election on $\mathcal{S}_5$

```
CODE FOR EACH PROCESS p:
 1: variables:
 2:     Leader_p ∈ {1,...,n}
 3:     SendTimer_p, ReceiveTimer_p: non-negative integers
 4:
 5: repeat forever
 6:     for all q ∈ V \ {p} do
 7:         if receive(ALIVE) from q then
 8:             if (Leader_p ≠ p) ∨ (q < p) then
 9:                 Leader_p ← q
10:             end if
11:             ReceiveTimer_p ← 0
12:         end if
13:     end for
14:     SendTimer_p ← SendTimer_p + 1
15:     if SendTimer_p ≥ ⌊δ/β⌋ then
16:         if Leader_p = p then
17:             send(ALIVE) to every process except p
18:         end if
19:         SendTimer_p ← 0
20:     end if
21:     ReceiveTimer_p ← ReceiveTimer_p + 1
22:     if ReceiveTimer_p > 8⌈δ/α⌉ then
23:         if Leader_p ≠ p then
24:             Leader_p ← p
25:         end if
26:         ReceiveTimer_p ← 0
27:     end if
28: end repeat
```

4 Self-Stabilizing Leader Election in $\mathcal{S}_4$

We first prove that we cannot implement any communication-efficient self-stabilizing leader election algorithm in $\mathcal{S}_2$ and $\mathcal{S}_4$. To that goal, we show that it is impossible to implement such an algorithm in a stronger system: $\mathcal{S}_5^-$ where $\mathcal{S}_5^-$ is any system $\mathcal{S}_0$ having (1) all its links that are reliable and (2) all its links that are timely except at most one which can be neither timely nor eventually timely.

Lemma 1. *Let $\mathcal{A}$ be any self-stabilizing leader election algorithm in $\mathcal{S}_5^-$. In any execution of $\mathcal{A}$, any alive process p satisfies: from any configuration where $Leader_p \neq p$, $\exists k \in \mathbb{R}^+$ such that p modifies $Leader_p$ if it receives no message during k times.*

Proof. Assume, by the contradiction, that there exists an execution e where there is a configuration γ from which a process p satisfies $Leader_p = q$ forever with $q \neq p$ while p does not receive a message anymore. As $\mathcal{A}$ is self-stabilizing, it can start from any configuration. So, $\overrightarrow{e_\gamma}$ is a possible execution. Let γ' be a configuration which is identical to γ except that q is crashed in γ'. Consider any execution $e_{\gamma'}$ starting from γ' where p did not receive a message anymore. As p cannot distinguish $\overrightarrow{e_\gamma}$ and $e_{\gamma'}$, it behaves in $e_{\gamma'}$ as in $\overrightarrow{e_\gamma}$: it keeps q as leader while q is crashed — a contradiction. □

Theorem 4. *There is no communication-efficient self-stabilizing leader election algorithm in system $\mathcal{S}_5^-$.*

Proof. Assume, by the contradiction, that there exists a communication-efficient self-stabilizing leader election algorithm $\mathcal{A}$ in system $\mathcal{S}_5^-$.

Consider any execution e where no process crashes and all the links behave as timely. By Definition 1 and Lemma 1, there exists a configuration γ in e such that in any suffix starting from γ: (1) there exists an alive process l such that any alive process p satisfies $Leader_p = l$ forever, and (2) messages are received infinitely often through at least one input link of each alive process except perhaps l.

Communication-efficiency and (2) implies that messages are received infinitely often in $\overrightarrow{e_\gamma}$ through exactly $n-1$ links of the form (q,p) with $p \neq l$. Let $E' \subset E$ be the subset containing the $n-1$ links where messages transit infinitely often in $\overrightarrow{e_\gamma}$.

Consider now any execution e' identical to e except that there is a time after which a certain link $(q,p) \in E'$ arbitrary delays the messages. (q,p) can behave as a timely link an arbitrary long time, so, e and e' can have an arbitrary large common prefix. In particular, e' can begin with any prefix of e of the form $\overleftarrow{e_\gamma} e''$ with e'' a non-empty prefix of $\overrightarrow{e_\gamma}$. Now, after any prefix $\overleftarrow{e_\gamma} e''$, (q,p) can start to arbitrary delay the messages and, in this case, p eventually changes its leader by Lemma 1. Hence, for any prefix $\overleftarrow{e_\gamma} e''$, there is a possible suffix of execution in $\mathcal{S}_5^-$ where p changes its leader: for some executions of $\mathcal{A}$ in $\mathcal{S}_5^-$ there is no guarantee that from a certain configuration the leader does not change anymore. Hence, $\mathcal{A}$ is not self-stabilizing in $\mathcal{S}_5^-$ — a contradiction. □

By definition, any system $\mathcal{S}_5^-$ is also a system $\mathcal{S}_2$ and any system $\mathcal{S}_5^-$ having $n \geq 3$ processes is a particular case of system $\mathcal{S}_4$. Hence:

Corollary 1. *There is no communication-efficient self-stabilizing leader election algorithm in systems $\mathcal{S}_2$ and $\mathcal{S}_4$ with $n \geq 3$ processes.*

Since $\mathcal{S}_4$ is a particular case of systems $\mathcal{S}_3$, Corollary 1 also holds for $\mathcal{S}_3$. However, a (non-communication-efficient) self-stabilizing leader election algorithm can be trivially implemented for $\mathcal{S}_3$, henceforth for $\mathcal{S}_4$ too, as explained afterwards. Any system $\mathcal{S}_3$ is characterized by the existence of a timely routing overlay. Using this characteristic, our solution works as follows: (1) every process p periodically sends an (ALIVE,1,p) message through all its links; (2) when receiving an (ALIVE,k,r) message from a process q, a process p retransmits an (ALIVE,$k+1$,r) message to all the other processes except q if $k < n-1$. Using this method, we have the guarantee that, any alive p periodically receives an (ALIVE,$-$,q) message for each other alive process q. Each process can then locally compute in an *Alives* set the list of all alive processes. Once the list is known by each alive process, designate a leader is easy: each alive process just outputs the smallest process of its *Alives* set.

5 Pseudo-Stabilizing Communication-Efficient Leader Election in $\mathcal{S}_4$

We now show that, contrary to self-stabilizing leader election, pseudo-stabilizing leader election can be communication-efficiently done in $\mathcal{S}_4$. To that goal, we

Algorithm 2. Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_4$

```
CODE FOR EACH PROCESS p:
 1: variables:
 2:     Leader_p ∈ {1,...,n}
 3:     SendTimer_p, ReceiveTimer_p, Round_p: non-negative integers
 4:
 5: procedure StartRound(s)
 6:     if p ≠ (s mod n + 1) then
 7:         send(START,s) to s mod n + 1
 8:     end if
 9:     Round_p ← s
10:     SendTimer_p ← ⌊δ/β⌋
11: end procedure
12:
13: repeat forever
14:     for all q ∈ V \ {p} do
15:         if receive (ALIVE,k) or (START,k) from q then
16:             if Round_p > k then
17:                 send(START,Round_p) to q
18:             else
19:                 if Round_p < k then
20:                     StartRound(k)
21:                 end if
22:                 ReceiveTimer_p ← 0
23:             end if
24:         end if
25:     end for
26:     ReceiveTimer_p ← ReceiveTimer_p + 1
27:     if ReceiveTimer_p > 8⌈δ/α⌉ then
28:         if p ≠ (Round_p mod n + 1) then
29:             StartRound(Round_p + 1)
30:         end if
31:         ReceiveTimer_p ← 0
32:     end if
33:     SendTimer_p ← SendTimer_p + 1
34:     if SendTimer_p ≥ ⌊δ/β⌋ then
35:         if p = (Round_p mod n + 1) then
36:             send(ALIVE,Round_p) to every process except p
37:         end if
38:         Leader_p ← (Round_p mod n + 1)
39:         SendTimer_p ← 0
40:     end if
41: end repeat
```

study an algorithm provided in [11]. In this algorithm (Algorithm 2), each process p executes in rounds $Round_p = 0, 1, 2, \ldots$, where the variable $Round_p$ keeps p's current round. For each round r a unique process, $l_r = r \bmod n + 1$, is distinguished: l_r is called the *leader of the round*. The goal here is to make all alive processes converge to a round value having an alive process as leader.

When starting a new round k, a process p (1) informs the leader of the round, l_k, by sending it a (START,k) message if $p \neq l_k$ (Line 6-8), (2) sets $Round_p$ to k (Line 9), and (3) forces $SendTimer_p$ to $\lfloor \delta/\beta \rfloor$ (Line 10) so that (a) p sends (ALIVE,k) to all other processes if $p = l_k$ (Lines 35-37) and (b) p updates $Leader_p$ (Line 38). While in the round r, the leader of the round l_r periodically sends (ALIVE,r) to all other processes (Lines 33-40). A process p modifies $Round_p$ only in two cases: (i) if p receives an ALIVE or START message with a round value bigger than its own (Lines 19-20), or (ii) if p does not recently receive an ALIVE message from its round leader $q \neq p$ (Lines 26-32). In case (i),

p adopts the round value in the message. In case (ii), p starts the next round (Line 29). Case (ii) allows a process to eventually choose as leader a process that correctly communicates. Case (i) allows the round values to converge. Intuitively, the algorithm is pseudo-stabilizing because, the processes with the upper values of rounds eventually designates as leader an alive process that correctly communicates forever (perhaps the bi-source) thanks to (ii) and, then, the other processes eventually adopt this leader thanks to (i).

Theorem 5. *Algorithm 2 implements a communication-efficient pseudo-stabilizing leader election in system $\mathcal{S}_4$.*

6 Impossibility of Self-Stabilizing Leader Election in $\mathcal{S}_2$

To prove that we cannot implement any self-stabilizing leader election algorithm in $\mathcal{S}_2$, we show that it is impossible to implement such an algorithm in a particular case of $\mathcal{S}_2$: let $\mathcal{S}_3^-$ be any system S_2 having all its links that are reliable but containing no eventually timely overlay.

Let m be any message sent at a given time t. We say that a message *m'* is *older* than m if and only if *m'* was initially in a link or *m'* was sent at a time t' such that $t' < t$. We call *causal sequence* any sequence p_0,m_1,...,m_i,p_i,m_{i+1},...,p_{k-1},m_k such that: (1) $\forall i$, $0 \leq i < k$, p_i is a process and m_{i+1} is a message, (2) $\forall i$, $1 \leq i < k$, p_i receives m_i from p_{i-1}, and (3) $\forall i$, $1 \leq i < k$, p_i sends m_{i+1} after the reception of m_i. By extension, we say that m_k *causally depends on* p_0. Also, we say that m_k is a *new* message that causally depends on p_0 after the message $m_{k'}$ if and only if there exists two causal sequences p_0,m_1,...,p_{k-1},m_k and p_0,$m_{1'}$,...,$p_{k'-1}$,$m_{k'}$ such that $m_{1'}$ is *older* than m_1.

Lemma 2. *Let $\mathcal{A}$ be any self-stabilizing leader election algorithm in S_3^-. In every execution of $\mathcal{A}$, any alive process p satisfies: from any configuration where $Leader_p \neq p$, $\exists k \in \mathbb{R}^+$ such that p changes its leader if it receives no* new *message that causally depends on $Leader_p$ during k times.*

Proof. Assume, by the contradiction, that there exists an execution e where there is a configuration γ from which a process satisfies $Leader_p = q$ forever with $q \neq p$ while from γ p does not receive anymore a *new* message that causally depends on q. As $\mathcal{A}$ is self-stabilizing, it can start from any configuration. So, $\overrightarrow{e_\gamma}$ is a possible execution of $\mathcal{A}$. Let γ' be a configuration that is identical to γ except that q is crashed in γ'. As p only received messages that do not causally depend on q in $\overrightarrow{e_\gamma}$ (otherwise, this means that from γ, p eventually receives at least one *new* message that causally depends on q in e), there exists a possible execution $\overrightarrow{e_{\gamma'}}$ starting from γ' where p received exactly the same messages as in $\overrightarrow{e_\gamma}$ (the fact that q is crashed just prevents p from receiving the messages that causally depend on q). Hence, p cannot distinguish $\overrightarrow{e_\gamma}$ and $\overrightarrow{e_{\gamma'}}$ and p behaves in $\overrightarrow{e_{\gamma'}}$ as in $\overrightarrow{e_\gamma}$: it keeps q as leader forever while q is crashed: $\mathcal{A}$ is not a self-stabilizing leader election algorithm — a contradiction. □

Theorem 6. *There is no self-stabilizing leader election algorithm in system S_3^-.*

Proof. Assume, by the contradiction, that there exists a self-stabilizing leader election algorithm $\mathcal{A}$ in system S_3^-. By Definition 1, in any execution of $\mathcal{A}$, there exists a configuration γ such that in any suffix starting from γ there exists a unique leader and this leader no more changes. Let e be an execution of $\mathcal{A}$ where no process crashes and every link is timely. Let l be the alive process which is eventually elected in e. Consider now any execution e' identical to e except that there is a time after which there is at least one link in each path from l to some process p that arbitrary delays messages. Then, e and e' can have an arbitrary large common prefix. Hence, we can construct executions of $\mathcal{A}$ beginning with any prefix of e where l is eventually elected (during this prefix, every link behaves as a timely link) but in the associated suffix, any causal sequence of messages from l to p is arbitrary delayed and, by Lemma 2, p eventually changes its leader to a process $q \neq l$. Thus, for any prefix $\overleftarrow{e}$ of e where a process is eventually elected, there exists a possible execution having $\overleftarrow{e}$ as prefix and an associated suffix $\overrightarrow{e}$ in which the leader eventually changes. Hence, for some executions of $\mathcal{A}$, we cannot guarantee that from a certain configuration the leader will no more change: $\mathcal{A}$ is not self-stabilizing — a contradiction. □

Intuitively, Theorem 6 means that self-stabilization is impossible for a weaker system than S_3, in particular, $\mathcal{S}_2$. Hence:

Corollary 2. *There is no self-stabilizing leader election algorithm in system $\mathcal{S}_2$.*

7 Communication-Efficient Pseudo-Stabilizing Leader Election in $\mathcal{S}_2$

From Corollary 2, we know that there does not exist any self-stabilizing leader election algorithm in $\mathcal{S}_2$. We now show that pseudo-stabilizing leader elections exist in $\mathcal{S}_2$. Furthermore we can achieve communication-efficiency. The solution we propose is an adaptation of an algorithm provided in [3].

To obtain communication-efficiency, Algorithm 3 uses the same principle as Algorithm 1: Each process p periodically sends ALIVE to all other processes *only if it thinks it is the leader*. However, this principle cannot be directly applied in $\mathcal{S}_2$: if the *only* source happens to be a process with a large ID, the leadership can oscillate among some other alive processes infinitely often because these processes can be alternatively considered as crashed or alive.

To fix the problem, Aguilera *et al* propose in [3] that each process p stores in an accusation counter, $Counter_p[p]$, how many time it was previously suspected to be crashed. Then, if p thinks that it is the leader, it periodically sends ALIVE messages with its current value of $Counter_p[p]$ (Lines 23-29). Any process stores in an $Actives$ set its own ID and that of each process it recently received an ALIVE message (Lines 8 and 12-16). Also, each process keeps the most up-to-date value of accusation counter of any process from which it receives an ALIVE message. Finally, any process q periodically chooses as leader the process having

Algorithm 3. Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_2$

```
CODE FOR EACH PROCESS p:
1: variables:
2:     Leader_p ∈ {1,...,n}, OldLeader_p ∈ {1,...,n}
3:     SendTimer_p, ReceiveTimer_p: non-negative integers
4:     Counter_p[1...n], Phase_p[1...n]: arrays of non-negative integers
5:     Collect_p, OtherActives_p: sets of non-negative integers
6:
7: macros:
8:     Actives_p = OtherActives_p ∪ {p}
9:
10: repeat forever
11:     for all q ∈ V \ {p} do
12:         if receive(ALIVE,qcnt,qph) from q then
13:             Collect_p ← Collect_p ∪ {q}
14:             Counter_p[q] ← qcnt
15:             Phase_p[q] ← qph
16:         end if
17:         if receive(ACCUSATION,ph) from q then
18:             if ph = Phase_p[p] then
19:                 Counter_p[p] ← Counter_p[p] + 1
20:             end if
21:         end if
22:     end for
23:     SendTimer_p ← SendTimer_p + 1
24:     if SendTimer_p ≥ ⌊δ/β⌋ then
25:         if Leader_p = p then
26:             send(ALIVE,Counter_p[p],Phase_p[p]) to every process except p
27:         end if
28:         SendTimer_p ← 0
29:     end if
30:     ReceiveTimer_p ← ReceiveTimer_p + 1
31:     if ReceiveTimer_p > 5⌈δ/α⌉ then
32:         OtherActives_p ← Collect_p
33:         if Leader_p ∉ Actives_p then
34:             send(ACCUSATION,Phase_p[Leader_p]) to Leader_p
35:         end if
36:         OldLeader_p ← Leader_p
37:         Leader_p ← r such that (Counter_p[r],r) = min{(Counter_p[q],q) : q ∈ Actives_p}
38:         if (OldLeader_p = p) ∧ (Leader_p ≠ p) then
39:             Phase_p[p] ← Phase_p[p] + 1
40:         end if
41:         Collect_p ← ∅
42:         ReceiveTimer_p ← 0
43:     end if
44: end repeat
```

the smallest accusation value among the processes in its $Actives_q$ set (IDs are used to break ties). After choosing a leader, if the leader of q changes, q sends an ACCUSATION message to its previous leader (Lines 33-35). The hope is that the counter of each source remains bounded, and, as a consequence, the source with the smallest counter is eventually elected.

However, the accusation counter of any source may increase infinitely often. Indeed, a source s can stop to consider itself as the leader: when s selects another process p as its leader. In this case, the source volontary stops sending ALIVE messages (for the communication efficiency), each other process that considered s as its leader eventually suspects s, and sends ACCUSATION messages to s. These messages cause incrementations of s'accusation counter. Later, due to the

quality of the output links of p, p can also increase its accusation counter and then the source may obtain the leadership again.

Aguilera *et al* add a mechanism so that a source increments its own accusation counter only a finite number of times. A process now increments its accusation counter only if it receives a "legitimate" accusation: an accusation due to the delay or the loss of one of its ALIVE message. To detect if an accusation is legitimate, each process p saves in $Phase_p[p]$ the number of times it loses the leadership in the past and includes this value in each of its ALIVE messages (Line 26). When a process q receives an ALIVE message from p, it also saves the phase value sent by p in $Phase_q[p]$ (Line 15). Hence, when q wants to accuse p, it now includes its own view of p's phase number in the ACCUSATION message it sends to p (Line 34). This ACCUSATION message will be considered as legitimate by p only if the phase number it contains matches the current phase value of p (Lines 18-20). Moreover, whenever p loses the leadership and stops sending ALIVE message voluntary, p increments $Phase_p[p]$ and does not send the new value to any other process (Line 38-40): this effectively causes p to ignore all the spurious ACCUSATION messages that result from its voluntary silence.

Theorem 7. *Algorithm 3 implements a communication-efficient pseudo-stabilizing leader election in system $\mathcal{S}_2$.*

8 Pseudo-Stabilizing Leader Election in $\mathcal{S}_1$

Let $\mathcal{S}_1^-$ be any system $\mathcal{S}_0$ with an eventually timely source and $n \geq 3$ processes. In [3], Aguilera *et al* show that there is no communication-efficient leader election algorithm in system $\mathcal{S}_1^-$. Now, any pseudo-stabilizing leader election algorithm in $\mathcal{S}_1$ is also a pseudo-stabilizing leader election algorithm in $\mathcal{S}_1^-$ by Theorem 2.

Theorem 8. *There is no communication-efficient pseudo-stabilizing leader election algorithm in system $\mathcal{S}_1$ with $n \geq 3$ processes.*

By Theorem 8, there is no communication-efficient pseudo-stabilizing leader election algorithm in system $\mathcal{S}_1$ with $n \geq 3$ processes. However, using similar techniques as those previously used in the paper, we can adapt the robust but non communication-efficient algorithm for $\mathcal{S}_1^-$ given in [**?**] to obtain a pseudo-stabilizing but non communication-efficient leader election algorithm for $\mathcal{S}_1$.

9 Future Works

There is some possible extensions to this work. First, getting a communication-efficient leader election in a system having a *timely routing overlay* remains an open question. Then, we can study robust stabilizing leader election in systems where only a given number of processes may crash. It could be interesting to extend these algorithms and results to other models like those in [18,12] and other communication topologies. Finally, we can study the implementability of robust stabilizing decision problems.

Acknowledgements. We are grateful to Ajoy K. Datta for his interesting remarks.

References

1. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. Technical report, LIAFA (2007) available at the following address, `http://hal.archives-ouvertes.fr/hal-00167935/fr/`
2. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery 17, 643–644 (1974)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: PODC 2003, pp. 306–314 (2003)
4. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC, pp. 195–206 (1993)
5. Anagnostou, E., Hadzilacos, V.: Tolerating transient and permanent failures (extended abstract). In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
6. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self-stabilization: Impossibility results and solutions using failure detectors. Int. J of Systems Science 28(11), 1177–1187 (1997)
7. Hutle, M., Widder, J.: Self-stabilizing failure detector algorithms. In: Parallel and Distributed Computing and Networks, pp. 485–490 (2005)
8. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: PODC, pp. 328–337 (2004)
9. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems 8(4), 424–440 (1997)
10. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: PODC 1999, pp. 199–207. ACM Press, New York (1999)
11. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
12. Malkhi, D., Oprea, F., Zhou, L.: Omega meets paxos: Leader election and stability without eventual timely links. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 199–213. Springer, Heidelberg (2005)
13. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
14. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distrib. Comput. 7(1), 35–42 (1993)
15. Larrea, M., Fernández, A., Arévalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: SRDS, pp. 52–59 (2000)
16. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with byzantine failures and little system synchrony. In: DSN, pp. 147–155 (2006)
17. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. journal version of [3](Unpublished)
18. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Brief announcement: Chasing the weakest system model for implementing omega and consensus. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 576–577. Springer, Heidelberg (2006)