Robust Stabilizing Leader Election

Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier

Robust Stabilizing Leader Election

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

August 23, 2007

Abstract

In this paper, we mix two well-known approaches of the fault-tolerance: *robustness* and *stabilization*. Robustness is the aptitude of an algorithm to withstand permanent failures such as process crashes. The stabilization is a general technique to design algorithms tolerating transient failures. Using these two approaches, we propose algorithms that tolerate both transient and crash failures. We study two notions of stabilization: the self- and the pseudo- stabilization (pseudo-stabilization is weaker than self-stabilization). We focus on the leader election problem. The goal here 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. In this work, we exhibit some assumptions required to obtain robust stabilizing leader election algorithms. Our results show, in particular, that the gap between robustness and stabilizing robustness is not really significant when we consider fix-point problems such as leader election.

Keywords: Distributed systems, self-stabilization, pseudo-stabilization, robust algorithm, leader election.

[1]LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France, {cd,hf}@liafa.jussieu.fr
[2]LaRIA, CNRS FRE 2733, Université de Picardie Jules Verne, 33 rue Saint Leu, 80000 Amiens, France, stephane.devismes@u-picardie.fr

1 Introduction

The quality of a modern distributed system mainly depends on its tolerance to the various kinds of faults that it may undergo. Two major kinds of faults are usually considered in the literature: the *transient* and *crash* failures. The *stabilization* introduced by Dijkstra in 1974 [12] is a general technique to design algorithms tolerating transient failures. In addition to the transient failures tolerance, the stabilization is highly desirable because, in many cases, stabilizing algorithms naturally (or with some minor modifications) withstand the dynamic topological changes. Finally, the initialization phase is not required in a stabilizing algorithm. Hence, stabilization is very interesting in dynamic and/or large-scale environments such as sensor networks and peer-to-peer systems. 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, *e.g.*, [3], tolerate the loss of messages which is a kind of transient failures). Actually, there is a few number of papers that deals with both stabilization and crash failures, *e.g.*, [14, 6, 20, 7, 9, 17, 16]. In [14], Gopal and Perry provide an algorithm that transforms fault-tolerant protocols into fault-tolerant self-stabilizing versions assuming a synchronous network. In [6], authors prove that *fault-tolerant self-stabilization* cannnot be achieve in asynchronous networks.

Here, we are interested in designing leader election algorithms that both tolerate transient and crash failures. Actually, we focus on finding stabilizing solutions in the message passing model with the possibility of some process crashes. The impossibility result of Aguilera *et al* ([4]) for robust leader election in asynchronous systems constraints us to make some assumptions on the link synchrony. So, we are looking 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 robust non-stabilizing systems (*e.g.* [2, 3]). In particular, it is also considered as a failure detector: eventually all alive processes agree on a common leader which is not crashed. Such a failure detector (called Ω) is important because it has been shown in [11] that it is the weakest failure detector with which one can solve the consensus.

The notion of stabilization appears in the literature with the well-known 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 [10], 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 (*i.e.*, a configuration from which the specification of the problem to solve is verified), it remains in a *legitimate* configuration forever (the *closure* property). In contrast, a pseudo-stabilizing algorithm just 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. There is some stabilizing non-robust leader election algorithms in the literature, *e.g.*, [13, 8].

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 [18]. 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 "equivalent" in (pseudo- or self-) stabilization in a sense that every algorithm which is stabilizing in a system $\mathcal{S}$ is also stabilizing in the system $\mathcal{S}'$ where $\mathcal{S}'$ is the same system as $\mathcal{S}$ except that all the synchronous links in $\mathcal{S}$ are eventually synchronous in $\mathcal{S}'$, and reciprocally. 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 a weaker system: any system $\mathcal{S}_3$ where there exists at least one path of synchronous links between each pair of alive processes. 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 some weak models: any system having a *timely bi-source*[1] ($\mathcal{S}_4$) 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 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

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

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

	$\mathcal{S}_5$	$\mathcal{S}_4$	$\mathcal{S}_3$	$\mathcal{S}_2$	$\mathcal{S}_1$	$\mathcal{S}_0$
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

Table 1: Implementability of the robust stabilizing leader election.

implemented in such systems. Finally, we conclude with the basic system where all links can be asynchronous and lossy ($\mathcal{S}_0$): it is clear that the leader election can be neither pseudo- nor self- stabilized in such a system. Table 1 summarizes our results.

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*

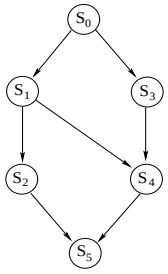

Figure 1: Systems considered in this paper ($\mathcal{S} \rightarrow \mathcal{S}'$ means $\mathcal{S}$ is weaker than $\mathcal{S}'$).

It is important to note that the solutions we propose are essentially adapted from previous existing robust algorithms provided, in particular, in [2, 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 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. Of course, adding a stabilizing property to robust algorithms allow to obtain algorithms that tolerate more types of failures: for example, the duplication and/or corruption of some messages.

Paper Outlines. In the following section, we present an informal model for our systems. We then consider the problem of the robust stabilizing leader election in various kinds of systems (Sections 3 to 10). Finally, we summarize our results and give some concluding remarks in Section 11.

2 Preliminaries

2.1 Distributed Systems

A distributed *system* is an aggregation of interconnected computing entities called *processes*. We consider here distributed systems where each process can communicate with each other through *directed links*: in the *communication network*, 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 and to share some resources. 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 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 $e = \gamma_0,\tau_0,\gamma_1,\tau_1,\ldots,\gamma_{i-1},\tau_{i-1},\gamma_i,\ldots$ 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 ($\tau_{i-1} \in \mathbb{R}$ and $\tau_{i-1} > 0$). 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 (*i.e.*, $e = \overleftarrow{e_\gamma}\overrightarrow{e_\gamma}$). Finally, we call *specification* a particular set of executions.

2.2 Self- and Pseudo- Stabilization

Formally, the self-stabilization can be defined as follows:

Definition 1 (Self-Stabilization [12]) *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 $\mathcal{F}$.*

Pseudo-stabilization is weaker than self-stabilization in a sense that any self-stabilizing algorithm is also a pseudo-stabilizing algorithm but the converse is not necessary true. Formally, the pseudo-stabilization can be defined as follows:

Definition 2 (Pseudo-Stabilization [10]) *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 $\mathcal{F}$.*

Self- versus Pseudo- Stabilization (from [10]). An algorithm $\mathcal{A}$ is self-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if starting from any arbitrary configuration, $\mathcal{A}$ guarantees that $\mathcal{S}$ reaches in a finite time a configuration from which $\mathcal{F}$ *cannot be* violated. In contrast, $\mathcal{A}$ is pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$ if and only if starting from any arbitrary configuration, $\mathcal{A}$ guarantees that $\mathcal{S}$ reaches in a finite time a configuration from which $\mathcal{F}$ *is not* violated. Thus, the only distinction between these two definitions comes down to the difference between "*cannot be*" and "*is not*". This difference may seem to be weak but actually is fondamental. In the case of self-stabilization, we have the guarantee that the system eventually reaches a configuration from which no deviation from $\mathcal{F}$ is possible. We have not such a guarantee with the pseudo-stabilization, we just know that the system eventually no more deviate from $\mathcal{F}$.

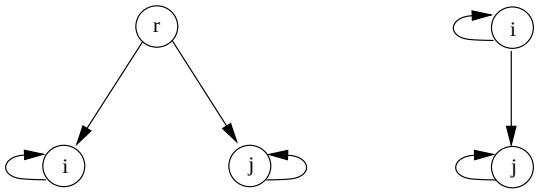

Figure 2: Self- and Pseudo-Stabilizing Algorithms.

Figure 2 illustrates the difference between these two properties. Consider the algorithm described by the state-transition diagram shown in Figure 2.(a) (in this diagram, circles represent configurations and oriented edges represent possible transitions). Starting from any configuration, the algorithm guarantees that the system reaches in at most one transition either the configuration i or the configuration j. From i (resp. j), only the execution (i,i,...) (resp. (j,j,...)) can be done. Thus, if the intended specification of the system is the set of executions $\mathcal{F} = \{(i,i,...), (j,j,...)\}$, then the system reaches within one transition a configuration (i or j) from which no deviation from $\mathcal{F}$ is possible. Hence, the algorithm is self-stabilizing for $\mathcal{F}$. Consider now the second algorithm provided in Figure 2.(b) and assume that the intended specification is still $\mathcal{F}$. The algorithm is not self-stabilizing because starting from i, it does not guarantee that the system will eventually leave i, now, in i the system can deviate from $\mathcal{F}$ if the algorithm executes (i,j,j,...) which is not in $\mathcal{F}$. On the other hand, every execution of the algorithm in the system is one of the following: (i,i,...), (i,...,i,j,j,...), or (j,j,...). Thus, every execution has an infinite suffix in $\mathcal{F}$. In other words, along every execution the algorithm guarantees that the system eventually reaches a configuration from which it does not deviate from $\mathcal{F}$, *i.e.*, the algorithm is pseudo-stabilizing for $\mathcal{F}$.

Robust Stabilization. Stabilization is a well-known technique allowing to design algorithms that tolerate *transient failures*. Roughly speaking, a transient failure is a temporary failure of some components of the system that can perturb its configuration. For instance, a transient failure can cause the corruption of some bits into some process memories or messages, as well as, the loss or the duplication of some messages. Actually, stabilizing algorithms withstand the transient failures because, after such failures, the system can be in an arbitrary configuration and, in this case, a stabilizing algorithm[3] guarantees that the system will recover a correct behavior in a finite time and without any external intervention if no transient failure appears during this convergence. To show the stabilization, we observe the system from the first configuration after the end of the last transient failure (yet considered as *the initial configuration* of system) and we assume that no more failure will occur. Actually, if we prove that from such a configuration and with such assumptions, an algorithm guarantees that the system recovers a correct behavior in a finite time, this means that this algorithm guarantees that the system will recover if the time between two periods of transient failures is sufficiently large. Henceforth, such an algorithm can be considered as tolerating transient failures.

In this paper, we not only consider the transient failures: our systems may go through transient as well as crash failures. Hence, our approach differs from the classical approach above presented. Here, 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.

[3] *n.b.*, in stabilization, its is usually assumed that the transient failures do not affect the code of the algorithms.

In the following, we will show that despite these constraints, it is possible (under some assumptions) to design (self- or pseudo-) 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. Hence, there exists a non-null lower bound on the time required by the alive processes to execute a step[4]. Moreover, every alive process is assumed to be *timely*, *i.e.*, 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 and we assume that each process can only execute a bounded number of steps in each loop iteration. Hence, each alive process satisfies a lower and an upper bound, respectively noted α and β, on the time it requires to execute an iteration of its *repeat forever* loop. We assume that α and β are known by each process.

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 automatically reset to $\perp$. Otherwise p does not receive any message in this step. In either case, p may change its state to reflect the outcome. Note that even if a message *m* of type T is delivered to p from q, there is no guarantee that p will eventually receive *m*. First, it is possible that p never chooses to check $\texttt{Buffer}_p[q,T]$. Second, it is also possible that $\texttt{Buffer}_p[q,T]$ is overwritten by a subsequent message from q of type T before p checks $\texttt{Buffer}_p[q,T]$ (however, in this case p receives *some* message of type T from q, but this is not *m*).

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 p having all its *output links* that are *timely* (resp. *eventually timely*). A *timely bi-source* (resp. an *eventually timely bi-source*) [5] is an alive process p 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

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

Theorems 1 and 2). Indeed, the finite period where the eventually timely links are asynchronous can be seen as a period of transient faults. Now, any stabilizing algorithm guarantees the convergence to a correct behavior after such a period.

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.

- **If.** 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 $\mathcal{S}$.
- **Only If.** Assume, by the contradiction, 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 $\overrightarrow{e_\gamma}$ (the suffix of e starting from γ) 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 of $\mathcal{S}$ can be an initial configuration) and (2) every eventually timely link of $\mathcal{S}'$ is timely in e_γ. 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. We said that an algorithm is *communication-efficient* [2] if there is a time from which it uses only $n-1$ unidirectional links.

Systems. We consider here six systems denoted by $\mathcal{S}_i$, $i \in [0...5]$. 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 (this system corresponds to the classical synchronous system). Figure 1 (page 2) summarizes the properties of our systems.

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. More formally, there exists an alive process l and a time t such that at any time $\forall t' \geq t$, every alive process p satisfies $Leader_p = l$.

A *robust pseudo-stabilizing leader election algorithm* guarantees that, starting from any configuration, the system reaches in a finite time a configuration γ from which any alive process p satisfies $Leader_p = l$ forever where l is an alive process.

A *robust self-stabilizing leader election algorithm* guarantees that, starting from any configuration, the system reaches in a finite time a configuration γ such that: (1) any alive process p satisfies $Leader_p = l$ in γ where l is an alive process and (2) any alive process p satisfies $Leader_p = l$ in any configuration reachable from γ.

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

We first seek a communication-efficient self-stabilizing leader election algorithm in a 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).

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        /* this ensures the convergence */
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        /* if p believes to be the leader, it periodically sends ALIVE to each other */
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        /* if ReceiveTimer_p expires and p does not believe to be the leader, */
23:        if Leader_p ≠ p then        /* p suspects its leader and, so, elects itself */
24:            Leader_p ← p
25:        end if
26:        ReceiveTimer_p ← 0
27:    end if
28: end repeat
```

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, p sets $Leader_p$ to q if $q < p$ (Lines 6-13). Using this mechanism, 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 below claims that, using the timestamps $\lfloor \delta/\beta \rfloor$ and $8\lceil \delta/\alpha \rceil$ respectively for $SendTimer_p$ and $ReceiveTimer_p$, Algorithm 1 implements a communication-efficient self-stabilizing leader election in any system $\mathcal{S}_5$. Due to the lack of space, the proof of Theorem 3 has been moved to the appendix (Section A, page 13).

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

4 Impossibility of Communication-Efficient Self-Stabilizing Leader Election in $\mathcal{S}_4$

To prove that we cannot implement any communication-efficient self-stabilizing leader election algorithm in $\mathcal{S}_4$, 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) having 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 $Leader_p \neq p$, $\exists k \in \mathbb{N}$ 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

Algorithm 2 Self-Stabilizing Leader Election on $\mathcal{S}_3$

CODE FOR EACH PROCESS p:

```
1: variables:
2:     Leader_p ∈ {1,...,n}
3:     SendTimer_p, ReceiveTimer_p: non-negative integers
4:     Collect_p, OtherAlives_p: sets of non-negative integers      /* these sets are used to compute the Alives_p set */
5:
6: macros:
7:     Alives_p = OtherAlives_p ∪ {p}
8:
9: repeat forever
10:    for all q ∈ V \ {p} do
11:        if receive(ALIVE,k,r) from q then
12:            Collect_p ← Collect_p ∪ {r}
13:            if k < n − 1 then
14:                send(ALIVE,k + 1,r) to every process except p and q      /* retransmission */
15:            end if
16:        end if
17:    end for
18:    SendTimer_p ← SendTimer_p + 1
19:    if SendTimer_p ≥ ⌊δ/β⌋ then      /* periodically p sends a new ALIVE message to every other process */
20:        send(ALIVE,1,p) to every process except p
21:        SendTimer_p ← 0
22:    end if
23:    ReceiveTimer_p ← ReceiveTimer_p + 1
24:    if ReceiveTimer_p > (4n − 3)⌈δ/α⌉ then      /* periodically, p selects a leader in Alives_p */
25:        OtherAlives_p ← Collect_p
26:        Leader_p ← min(Alives_p)
27:        Collect_p ← ∅
28:        ReceiveAliveTimer_p ← 0
29:    end if
30: end repeat
```

that q is crashed in γ'. Consider then 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 any system $\mathcal{S}_5^-$.*

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

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

Let $\overrightarrow{e_\gamma}$ be the suffix of e where every alive process p satisfies $Leader_p = l$ forever. 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 changes anymore. Hence, $\mathcal{A}$ is not self-stabilizing in $\mathcal{S}_5^-$ — a contradiction. □

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

Corollary 1 *There is no communication-efficient self-stabilizing leader election algorithm in a system $\mathcal{S}_4$ having $n \geq 3$ processes.*

5 Self-Stabilizing Leader Election in $\mathcal{S}_3$ and $\mathcal{S}_4$

$\mathcal{S}_4$ is a particular case of systems $\mathcal{S}_3$. So, by Corollary 1, there does not exist any self-stabilizing communication-efficient leader election algorithm working in any system $\mathcal{S}_3$ or $\mathcal{S}_4$. We now present a non-communication-efficient self-stabilizing leader election algorithm for $\mathcal{S}_3$: Algorithm 2. By definition, this algorithm is also self-stabilizing in $\mathcal{S}_4$. However, using the characterics of $\mathcal{S}_4$, it can be simplified for working in $\mathcal{S}_4$ as explained at the end of the section.

Algorithm 2 consists in locally computing in the set $Alives$ 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.

Any system $\mathcal{S}_3$ is characterized by the existence of a timely routing overlay, *i.e.*, for each pair of alive processes (p,q) there exists at least two elementary paths of timely links: one from p to q and the other from q to p. Using this characteristic, our algorithm works as follows: (1) every process p periodically sends an (ALIVE,1,p) message through all its links (Line 20 of Algorithm 2); (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 $k < n-1$ (Lines 13-15).

Using this method, we have the guarantee that, any alive p periodically receives an (ALIVE,$-$,q) message for each other alive process q. Indeed, as there exists a timely routing overlay in the system, for each pair of alive processes (p,q), there exists at least one elementary path of timely links from q to p whose length is bounded by $n-1$ (the upper bound on the diameter of the timely routing overlay), and conversely. Hence, each process p can periodically compute a $Collect_p$ set where it stores the IDs of every other alive process: the IDs contained in all the messages it recently received. Eventually, the IDs of every crashed process does not appear in the $Collect$ sets anymore. Moreover, the timely routing overlay guarantees that the IDs of each other alive process are periodically assigned into the $Collect$ sets of all alive processes. Hence, by periodically assigning the content of $Collect_p$ (using a period sufficiently large) to the set $OtherAlives_p$ (Line 25), we can guarantee the convergence of $OtherAlives_p$ to the set of all the alive processes different of p. Finally, p just has to periodically choose its leader in the set $Alives_p = OtherAlives_p \cup \{p\}$ (Line 26) so that the system eventually converges to a unique leader. Finally, note that Algorithm 2 still uses one message type: ALIVE, and the two counters: $SendTimer_p$ and $ReceiveTimer_p$.

Theorem 5 below claims that, using the timestamps $\lfloor \delta/\beta \rfloor$ and $(4n-3)\lceil \delta/\alpha \rceil$ respectively for $SendTimer_p$ and $ReceiveTimer_p$, Algorithm 2 is self-stabilizing for the leader election problem in any system $\mathcal{S}_3$. The proof of Theorem 5 is provided in the appendix (Section B, page 16).

Theorem 5 *Algorithm 2 implements a self-stabilizing leader election in System $\mathcal{S}_3$.*

$\mathcal{S}_4$ is a particular case of $\mathcal{S}_3$. Indeed, there exists a timely routing overlay in any system $\mathcal{S}_4$ due to the existence of a bi-source. But, in $\mathcal{S}_4$, the diameter of the timely routing overlay is bounded by 2 instead of $n-1$ in $\mathcal{S}_3$. Hence, the ALIVE messages need to be repeated only once in $\mathcal{S}_4$ to get the guarantee that each alive process receives them in a bounded amount of time. Hence, Algorithm 2 remains self-stabilizing in any system $\mathcal{S}_4$ if we replace the timestamp of $ReceiveTimer_p$ by $9\lceil \delta/\beta \rceil$ (*i.e.*, $(4d+1)\lceil \delta/\beta \rceil$ with the diameter $d=2$) and the test of Line 13 by the test "$k < 2$".

6 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 study an algorithm provided in [2]. In this algorithm, 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 a unique process, $q = Round_p \bmod n+1$, is distinguished: q 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 $\lceil \delta/\alpha \rceil$ (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 ($l_r = r \bmod n + 1$) 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). Finally, note that Algorithm 3 uses two message types: ALIVE and START and the two counters: $SendTimer_p$ and $ReceiveTimer_p$.

Theorem 6 below claims that, using the timestamps $\lfloor \delta/\beta \rfloor$ and $8\lceil \delta/\alpha \rceil$ respectively for $SendTimer_p$ and $ReceiveTimer_p$, Algorithm 3 is pseudo-stabilizing and communication-efficient for the leader election problem in any system $\mathcal{S}_5$. The proof of Theorem 6 is given in the appendix (Section C, page 17).

Theorem 6 *Algorithm 3 implements a communication-efficient pseudo-stabilizing leader election in System $\mathcal{S}_4$.*

7 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

Algorithm 3 Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_4$

CODE FOR EACH PROCESS p:

1: **variables:**
2: $Leader_p \in \{1,...,n\}$
3: $SendTimer_p$, $ReceiveTimer_p$, $Round_p$: non-negative integers
4:
5: **procedure** $StartRound(s)$ /* this procedure is called each time p increases its round value */
6: **if** $p \neq (s \bmod n + 1)$ **then**
7: send(START,s) to $s \bmod n + 1$
8: **end if**
9: $Round_p \leftarrow s$
10: $SendTimer_p \leftarrow \lfloor \delta/\beta \rfloor$
11: **end procedure**
12:
13: **repeat forever**
14: **for all** $q \in V \setminus \{p\}$ **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** /* to ensure the convergence */
20: $StartRound(k)$
21: **end if**
22: $ReceiveTimer_p \leftarrow 0$ /* if $k \geq Round_p$, p restarts $ReceiveTimer_p$ */
23: **end if**
24: **end if**
25: **end for**
26: $ReceiveTimer_p \leftarrow ReceiveTimer_p + 1$
27: **if** $ReceiveTimer_p > 8\lceil \delta/\alpha \rceil$ **then** /* on time out p changes its round value if p is not the leader of current round */
28: **if** $p \neq (Round_p \bmod n + 1)$ **then**
29: $StartRound(Round_p + 1)$
30: **end if**
31: $ReceiveTimer_p \leftarrow 0$
32: **end if**
33: $SendTimer_p \leftarrow SendTimer_p + 1$
34: **if** $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ **then**
35: **if** $p = (Round_p \bmod n + 1)$ **then**
36: send(ALIVE,$Round_p$) to every process except p /* the leader of the round periodically send ALIVE to each other process */
37: **end if**
38: $Leader_p \leftarrow (Round_p \bmod n + 1)$ /* p periodically computes $Leader_p$ */
39: $SendTimer_p \leftarrow 0$
40: **end if**
41: **end repeat**

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 $Leader_p \neq p$, $\exists k \in \mathbb{N}$ 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 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 7 *There is no self-stabilizing leader election algorithm in a system S_3^-.*

Proof. Assume, by the contradiction, that there exists a self-stabilizing leader election algorithm $\mathcal{A}$ in a 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 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, it is possible to 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 a self-stabilizing leader election algorithm — a contradiction. □

By Definition, any system S_3^- is also a system $\mathcal{S}_2$. Hence, follows:

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

8 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 $\mathcal{S}_2$. The solution we propose is an adaptation of an algorithm provided in [3] and is communication-efficient.

Algorithm 4 Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_2$

CODE FOR EACH PROCESS p:

1: **variables:**
2: $Leader_p \in \{1,...,n\}, OldLeader_p \in \{1,...,n\}$
3: $SendTimer_p, ReceiveTimer_p$: non-negative integers
4: $Counter_p[1...n], Phase_p[1...n]$: arrays of non-negative integers /* to manage the accusations */
5: $Collect_p, OtherActives_p$: sets of non-negative integers /* these sets are used to compute the $Actives_p$ set */
6:
7: **macros:**
8: $Actives_p = OtherActives_p \cup \{p\}$
9:
10: **repeat forever**
11: **for all** $q \in V \setminus \{p\}$ **do**
12: **if** receive(ALIVE,$qcnt$,qph) from q **then** /* $qcnt$ and qph correspond to the value of $Counter_q[q]$ and $Phase_q[q]$ when q sends the message */
13: $Collect_p \leftarrow Collect_p \cup \{q\}$
14: $Counter_p[q] \leftarrow qcnt$
15: $Phase_p[q] \leftarrow qph$
16: **end if**
17: **if** receive(ACCUSATION,ph) from q **then** /* on reception of an ACCUSATION message */
18: **if** $ph = Phase_p[p]$ **then** /* if the accusation is legitimate */
19: $Counter_p[p] \leftarrow Counter_p[p] + 1$ /* $Counter_p[p]$ is incremented */
20: **end if**
21: **end if**
22: **end for**
23: $SendTimer_p \leftarrow SendTimer_p + 1$
24: **if** $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ **then** /* if p believes to be the leader, it periodically sends ALIVE to each other */
25: **if** $Leader_p = p$ **then**
26: send(ALIVE,$Counter_p[p]$,$Phase_p[p]$) to every process except p
27: **end if**
28: $SendTimer_p \leftarrow 0$
29: **end if**
30: $ReceiveTimer_p \leftarrow ReceiveTimer_p + 1$
31: **if** $ReceiveTimer_p > 5\lceil \delta/\alpha \rceil$ **then**
32: $OtherActives_p \leftarrow Collect_p$
33: **if** $Leader_p \notin Actives_p$ **then** /* p sends an ACCUSATION message to its leader when it suspects it */
34: send(ACCUSATION,$Phase_p[Leader_p]$) to $Leader_p$
35: **end if**
36: $OldLeader_p \leftarrow Leader_p$
37: $Leader_p \leftarrow r$ such that $(Counter_p[r],r) = \min\{(Counter_p[q],q) : q \in Actives_p\}$ /* p periodically computes $Leader_p$ */
38: **if** $(OldLeader_p = p) \wedge (Leader_p \neq p)$ **then** /* when p loses its leadership, it increments its phase */
39: $Phase_p[p] \leftarrow Phase_p[p] + 1$
40: **end if**
41: $Collect_p \leftarrow \emptyset$
42: $ReceiveTimer_p \leftarrow 0$
43: **end if**
44: **end repeat**

To obtain communication-efficiency, Algorithm 4 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 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 (because all its output links are timely), and, as a consequence, the source with the smallest counter is eventually elected.

However, this algorithm still does not work in $\mathcal{S}_2$: the accusation counter of any source may increase infinitely often. To see this, note that a source s can stop to consider itself as the leader: when s selects another process p as its leader (a process in $Actives_s$ with a smaller counter). In this case, the source volontary stops sending ALIVE messages for the communication efficiency. Unfortunately, each other process that considered s as its leader eventually suspects s and, so, sends ACCUSATION messages to s. These messages then cause incrementations of s'accusation counter. Later, due to the quality of the output links of p (p may not be a source), p can also increase its accusation counter and then the source may obtain the leadership again. As a consequence, the leadership may oscillate infinitely often.

To guarantee that the leadership does not oscillate infinitely often, Aguilera *et al* add a mechanism so that the 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 and not due to the fact that it voluntary stopped sending messages. 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. Finally, note that Algorithm 4 uses two message types: ALIVE and ACCUSATION, as well as, the two counters: $SendTimer_p$ and $ReceiveTimer_p$.

Theorem 8 below claims that, using the timestamps $\lfloor \delta/\beta \rfloor$ and $5\lceil \delta/\alpha \rceil$ respectively for $SendTimer_p$ and $ReceiveTimer_p$, Algorithm 4 is pseudo-stabilizing and communication-efficient for the leader election problem in any system $\mathcal{S}_2$. Due to the lack of space, the proof of Theorem 8 has been moved to the appendix (Section D, page 19).

Theorem 8 *Algorithm 4 implements a communication-efficient pseudo-stabilizing leader election in System $\mathcal{S}_2$.*

9 Impossibility of Communication-Efficient 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 a 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 (page 5). Hence, follows:

Theorem 9 *There is no communication-efficient pseudo-stabilizing leader election algorithm in a system $\mathcal{S}_1$ having $n \geq 3$ processes.*

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

By Theorem 9, there is no communication-efficient pseudo-stabilizing leader election algorithm in a system $\mathcal{S}_1$ having $n \geq 3$ 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 [1] to obtain a pseudo-stabilizing but non communication-efficient leader election algorithm for $\mathcal{S}_1$. Due to the lack of space, we do not present the algorithm here, but the algorithm and its proof of pseudo-stabilization are provided in the appendix (Section E, page 22).

11 Conclusion and Future Works

We studied the problem of implementing robust self- and pseudo- stabilizing leader election in various systems with weak reliability and synchrony assumptions. We tried to propose, when it is possible, *communication-efficient* implementations. We first show that the notions of immediate timeliness and eventually timeliness are "equivalent" in

stabilization in a sense that every algorithm which is stabilizing 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. Hence, we only consider timely properties that are immediate. We study systems where (1) all the processes are timely and can communicate with each other but some of them may crash and, (2) some links may have timely and reliability properties. We first showed that the *full timeliness* is minimal to have any self-stabilizing communication-efficient leader election in the systems we consider. Nevertheless, we showed that a self-stabilizing leader election that is not communication-efficient can be obtained in a weaker system: a system where there exists *a timely routing overlay*. We also showed that no self-stabilizing leader election can be implemented in our systems without this assumption. Hence, we then focused on the pseudo-stabilization. We showed that leader election can be communication-efficiently pseudo-stabilized in the same systems than those where robust leader elections exist: in systems having a *timely bi-source* and systems having a *timely source* and *fair* links (note that getting communication-efficiency in a system having a *timely routing overlay* remains an open question). Using then a previous result of Aguilera *et al* ([3]), we recalled that communication-efficiency cannot be done if we consider systems having at least *one timely source* but where the fairness of all the links is not required. Finally, we showed that, as the robust leader election, the pseudo-stabilizing leader election can be non-communication-efficiently implemented in such systems. Hence, we can have a robust pseudo-stabilizing leader election in almost all the systems where a robust leader election already exists: the gap between robustness and pseudo-stabilizing robustness is not really significant in fix-point problems such as leader election.

There is some possible extensions to this work. First, we can study robust stabilizing leader election in systems where only a given number of processes may crash. Then, we can consider the robust stabilizing leader election in some other models as those in [15, 19]. We can also consider the robust stabilizing leader election in systems with various topology. Finally, we can study the implementability of robust stabilizing decision problems such as *consensus*.

References

[1] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. Unpublished, journal version of [3], January 2007.

[2] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *DISC*, pages 108–122, 2001.

[3] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC*, pages 306–314, 2003.

[4] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, pages 328–337, 2004.

[5] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In *DSN*, pages 147–155, 2006.

[6] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *WDAG*, pages 174–188, 1993.

[7] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: Impossibility results and solutions using failure detectors. *Int. J of Systems Science*, (11):1177–1187, 1997.

[8] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 199–207, New York, NY, USA, 1999. ACM Press.

[9] Joffroy Beauquier and Synnöve Kekkonen-Moneta. On ftss-solvable distributed problems. In *PODC*, page 290, 1997.

[10] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distrib. Comput.*, 7(1):35–42, 1993.

[11] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[12] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[13] S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[14] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *PODC*, pages 195–206, 1993.

[15] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Brief announcement: Chasing the weakest system model for implementing omega and consensus. In *Proceedings Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, LNCS, pages 576–577, Dallas, USA, Nov. 2006. Springer Verlag.

[16] Martin Hutle and Josef Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Self-Stabilizing Systems*, pages 153–170, 2005.

[17] Martin Hutle and Josef Widder. Self-stabilizing failure detector algorithms. In *Parallel and Distributed Computing and Networks*, pages 485–490, 2005.

[18] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS*, pages 52–59, 2000.

[19] Dahlia Malkhi, Florian Oprea, and Lidong Zhou. *mega* meets paxos: Leader election and stability without eventual timely links. In *DISC*, pages 199–213, 2005.

[20] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. *Proceedings of the second Workshop on Self-Stabilizing Systems*, page article 1, 1995.

APPENDIX

The following observation is used along the proofs of Theorems A to E.

Observation 1 *For every alive process p, for every time t, p executes at least one* **complete** *iteration of its* repeat forever *loop during the time interval $[t, t + 2\beta[$.*

A Proof of Theorem 3

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), we are sure that any process p sends a message only if the test of Line 16 is true, *i.e.*, only if $Leader_p = p$ [5]. Hence:

Observation 2 *Starting from any configuration, a process p sends a message at time $t{>}\beta$ only if $Leader_p{=}p$ at time t.*

Lemma 3 *Starting from any configuration, if a process q receives a message* m *at time $t > \delta + 3\beta$, then there exists another alive process p that sends* m *while $Leader_p = p$ at a time t' such that $t - (\delta + 2\beta) \leq t' < t$.*

Proof. The lemma is proven by the following three claims:

1. *Any process that is crashed in the initial configuration never sends any message during the execution.*

2. *q cannot receive at time $t > \delta + 2\beta$ a message that was in a link since the initial configuration.*

   Claim Proof: In $\mathcal{S}_5$, all messages initially in the links are delivered at most at time δ. When q receives such a message, it is received at most one complete *repeat forever* loop iteration after its delivrance: at most at time $\delta + 2\beta$ by Observation 1. So, any message received by q at any time $t > \delta + 2\beta$ was not initially in the link.

3. *q receives a message* m *from the alive process p at time $t > \delta + 3\beta$ only if p sends* m *while satisfying $Leader_p = p$ at a time t' such that $t - (\delta + 2\beta) \leq t' < t$.*

   Claim Proof: By Claim 2, q receives *m* at time $t > \delta + 3\beta$ only if p effectively sends *m* to q at a time $t' < t$. As q receives *m* at most 2β times (one complete iteration of the *repeat forever* loop) after its delivrance and *m* is delivered at most δ times after its sending, we can deduce that $t' \geq t - (\delta + 2\beta)$. Finally, as $t' \geq t - (\delta + 2\beta)$ and $t > \delta + 3\beta$, we have $t' > \beta$ and, by Observation 2, we can deduce that p satisfies $Leader_p = p$ at time t'.

□

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), any process q sets $Leader_q$ to $p \neq q$ only if q previously receives ALIVE from p. Hence, follows:

Observation 3 *Starting from any configuration, any process q sets $Leader_q$ to $p \neq q$ at time $t > \beta$ only if q previously receives ALIVE from p.*

From the code of Algorithm 1, Observation 3, and Lemma 3, we can deduce the following lemma:

Lemma 4 *Starting from any configuration, any process q switches $Leader_q$ from q to $p \neq q$ at time $t > \delta + 3\beta$ only if: (1) p is an alive process and $p{<}q$, and (2) p sends ALIVE to q while $Leader_p{=}p$ at a time t' with $t{-}(\delta{+}2\beta){\leq}\ t'{<}t$.*

Definition 3 *Let $Candidates(t)$ be the set containing any alive process p such that $Leader_p = p$ at time t.*

Lemma 5 *Starting from any configuration, $\forall i \in \mathbb{N}^+$, $\forall t > \beta + i(\delta + 2\beta)$, if $Candidates(t) > 0$ and $\exists t' > t$ such that $Candidates(t') = 0$, then there exists an alive process p such that $p < [min(Candidates(t)) - (i - 1)]$ and a time t_i with $t - i(\delta + 2\beta) < t_i < t'$ such that $Leader_p = p$ at time t_i.*

Proof. By induction on i.

Induction for $i = 1$: Let t be a time such that $t > \delta + 3\beta$. Assume that $Candidates(t) > 0$ and $\exists t' > t$ such that $Candidates(t') = 0$. Let $q = \min(Candidates(t))$. There is a time t_j such that $t < t_j \leq t'$ where q switches $Leader_q$ from q to $p \neq q$. By Lemma 4, p is an alive process such that $p < q$ and p sends ALIVE to q while $Leader_p = p$ at a time t_i with $t_j - (\delta + 2\beta) \leq t_i < t_j$. Now, $t < t_j \leq t'$. So, $t - (\delta + 2\beta) < t_i < t'$ and the induction holds for $i = 1$.

[5] *n.b.*, the program counter of p can initially point out to Line 17: then p may send messages during the first loop iteration while $Leader_p \neq p$.

Induction Assumption: Let $k \in \mathbb{N}^+$. Assume that $\forall i \in \mathbb{N}^+$ such that $i \leq k$ we have: $\forall t > \beta + i(\delta + 2\beta)$, if $Candidates(t) > 0$ and $\exists t' > t$ such that $Candidates(t') = 0$, then there exists an alive process $p < [min(Candidates(t)) - (i-1)]$ and a time t_i with $t - i(\delta + 2\beta) < t_i < t'$ such that $Leader_p = p$ at time t_i.

Induction for $i = k + 1$: Let t be a time such that $t > \beta + (k+1) \times (\delta + 2\beta)$. Assume that $Candidates(t) > 0$ and $\exists t' > t$ such that $Candidates(t') = 0$. Let $q = \min(Candidates(t))$. As previously, there is a time t_j such that $t < t_j \leq t'$ where q switches $Leader_q$ from q to $r \neq q$ and, by Lemma 4, r is an alive process such that $r < q$ and r sends ALIVE to q while $Leader_r = r$ at a time t_r with $t_j - (\delta + 2\beta) \leq t_r < t_j$. Now, $t_r > \beta + k \times (\delta + 2\beta)$ and $Candidates(t_r) > 0$, so, by induction assumption: there exists an alive process $p < min(Candidates(t_r)) - (k-1)$ and a time t_k with $t_r - k(\delta + 2\beta) < t_k < t'$ such that $Leader_p = p$ at time t_k.

(a) We now show that $p < [min(Candidates(t)) - k]$. First, $min(Candidates(t_r)) \leq r$, so, $p < r - (k-1)$. Then, $r < q$, so, $r \leq q-1$ (remember that $V = \{1,...,n\}$). Hence, $p < q-1-(k-1)$, *i.e.*, $p < [min(Candidates(t) - k]$.

(b) Finally, we show that p is an alive process such that $Leader_p = p$ at time t_k with $t - (k+1) \times (\delta + 2\beta) < t_k < t'$. First, we already know that p is an alive process such that $Leader_p = p$ at time t_k. Then, $t < t_j$ and $t_j - (\delta + 2\beta) \leq t_r$ implies that $t - (\delta + 2\beta) < t_r$. Finally, as $t_r - k(\delta + 2\beta) < t_k < t'$ and $t - (\delta + 2\beta) < t_r$, we have $[t - (\delta + 2\beta) - k(\delta + 2\beta)] < t_k < t'$. Hence, p satisfies $Leader_p = p$ at time t_k with $t - (k+1) \times (\delta + 2\beta) < t_k < t'$.

Hence, by (a) and (b), we can deduce that the induction holds for $i = k + 1$. □

Lemma 6 *Starting from any configuration,* $\forall t > \beta + n(\delta + 2\beta)$, $(Candidates(t) > 0) \Rightarrow (Candidates(t') > 0, \forall t' > t)$.

Proof. Assume, by the contradiction, that $\exists t > \beta + n(\delta + 2\beta)$ such that $Candidates(t) > 0$ and $\exists t' > t$ such that $Candidates(t') = 0$. Then, by Lemma 5, there exists an alive process p such that $p < min(Candidates(t)) - (n-1)$ and a time t'' with $t - n(\delta + 2\beta) < t'' < t'$ such that $Leader_p = p$ at time t''. Now, $min(Candidates(t)) \leq n$ ($V = \{1,...,n\}$). So, $p < n - (n-1)$, *i.e.*, $p < 1$ — a contradiction. □

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), any process p executes Line 11 of the algorithm only if the test of Line 7 is true. Hence, follows:

Observation 4 *Starting from any configuration, any process p executes Line 11 at time $t > \beta$ only if p previously receives an ALIVE message (in the same iteration of the* repeat forever *loop).*

Lemma 7 *Starting from any configuration,* $\forall t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$, $Candidates(t) > 0$.

Proof. Consider the time interval $[(n+1)(\delta + 2\beta) + 2\beta + 1, (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1]$.

- Assume that there exists a process p that executes Line 11 at a time $t \in [(n+1)(\delta + 2\beta) + 2\beta + 1, (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1]$. Then, p receives an ALIVE message from a process q before executing Line 11 but in the same iteration of the *repeat forever* loop by Observation 4, *i.e.*, at most β times before. So, p receives an ALIVE message from q at a time $t' \in [(n+1)(\delta + 2\beta) + \beta + 1, (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1[$. By Lemma 3, q is alive and sends ALIVE while satisfying $Leader_q = q$ at a time t'' such that $t' - (\delta + 2\beta) \leq t'' < t'$. So, $Candidates(t'') > 0$ with $t'' \in [n(\delta + 2\beta) + \beta + 1, (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1[$ and $\forall t''' > t''$, $Candidates(t''') > 0$ by Lemma 6. As $t'' < (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$, the lemma holds in this case.

- Assume that no process executes Line 11 during the time interval $[(n+1)(\delta + 2\beta) + 2\beta + 1, (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1]$.

  (i) If $Candidates((n+1)(\delta + 2\beta) + 2\beta + 1) > 0$, then $\forall t > (n+1)(\delta + 2\beta) + 2\beta + 1$, $Candidates(t) > 0$ by Lemma 6 and the lemma holds in this case.

  (ii) Assume now that $Candidates((n+1)(\delta + 2\beta) + 2\beta + 1) = 0$, *i.e.*, any alive process p satisfies $Leader_p \neq p$ at time $(n+1)(\delta + 2\beta) + 2\beta + 1$. Then, the program counter of any alive process p points out to the first instruction of the *repeat forever* loop at a time $(n+1)(\delta + 2\beta) + 2\beta + 1 \leq t \leq (n+1)(\delta + 2\beta) + 3\beta + 1$. From t, p executes a complete iteration of the loop at most every β times. So, each p executes at least $8\lceil \delta/\alpha \rceil + 1$ complete loop iterations from time t to time $(n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$. Now, we assume that no process executes Line 11 from time t to time $(n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$. So, during this period, we are sure that, for each alive process p, $ReceiveTimer_p$ is incremented at each loop iteration until $ReceiveTimer_p > 8\lceil \delta/\alpha \rceil$. As $ReceiveTimer$ is always greater or equal to 0, any alive process satisfies $ReceiveTimer_q > 8\lceil \delta/\alpha \rceil$ and sets $Leader_p$ to p at the lattest during the $(8\lceil \delta/\alpha \rceil + 1)^{th}$ loop iteration executed in the time interval we consider. Thus, any p sets $Leader_p$ to p at a time $t' \leq (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$. In this case, $Candidates(t') > 0$ and the lemma holds by Lemma 6.

□

Lemma 8 *Starting from any configuration, if an alive process p continuously satisfies $Leader_p = p$ during the time interval $[t, t + \delta + \beta]$, then p sends at least one ALIVE message to any other process during this time interval.*

Proof. Let t be any time. From t, the program counter of p points out to the first instruction of the *repeat forever* loop at a time $t' \leq t + \beta$. From t', p executes a complete iteration of the loop at most every β times. Also, from t', while $SendTimer_p < \lfloor \delta/\beta \rfloor$, $SendTimer_p$ is incremented at each loop iteration. So, as $SendTimer_p$ is always greater or equal to 0, $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ becomes true at the lattest during the $\lfloor \delta/\beta \rfloor^{th}$ loop iteration from t' and p sends ALIVE to any other process in the same loop iteration (Lines 14-20). Hence, from t', p sends ALIVE to any other process in at most $\lfloor \delta/\beta \rfloor \times \beta$ times, *i.e.*, in at most δ times. As $t' \leq t + \beta$, the lemma is proven. □

Lemma 9 *Starting from any configuration, $\forall t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$, $\exists t' \in [t, t + 2\delta + 3\beta]$ such that an alive process sends ALIVE to every other processes at time t'.*

Proof. Let t such that $t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$. By Lemma 7, $\forall t'$, $t' \geq t$, there exists at least one alive process q such that $Leader_q = q$ at time t'. Let p be an alive process such that $Leader_p = p$ at time $t + \delta + 2\beta$.

- Assume that p continuously satisfies $Leader_p = p$ during the time interval $[t + \delta + 2\beta, t + 2\delta + 3\beta]$. Then, p sends at least one ALIVE message to any other process during $[t + \delta + 2\beta, t + 2\delta + 3\beta]$ by Lemma 8.

- Assume that there is a time $t' \in]t+\delta+2\beta, t+2\delta+3\beta]$ where p sets $Leader_p$ to q such that $q \neq p$. Then, q is alive and q sends ALIVE to p at a time t'' such that $t' - (\delta + 2\beta) \leq t'' < t'$ by Lemma 4. From Algorithm 1, q sends ALIVE to every other process at time t''. Finally, as $t+\delta+2\beta < t' \leq t+2\delta+3\beta$, we have $t < t'' \leq t+2\delta+3\beta$. Hence, at least one alive process (actually, q) sends ALIVE to every other processes during $[t, t + 2\delta + 3\beta]$.

□

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), we are sure that a process p sets $Leader_p$ to p (Line 24) only if the two tests of Lines 22-23 are true. Hence, follows:

Observation 5 *Starting from any configuration, any process p sets $Leader_p$ to p at time $t > \beta$ only if $(Leader_p \neq p) \wedge (ReceiveTimer_p > 8\lceil \delta/\alpha \rceil)$ at time t.*

Lemma 10 *Starting from any configuration, p sets $Leader_p$ to p at time $t > 8\delta$ only if p do not receive any ALIVE message during $[t - 8\delta, t]$.*

Proof. Assume, by the contradiction, that an alive process p receives at least one ALIVE message during $[t - 8\delta, t]$ (with $t > 8\delta$) but p sets $Leader_p$ to p at time t. From Algorithm 1, after receiving ALIVE (Line 7), p resets $ReceiveTimer_p$ to 0 (Line 11) and p does not set $Leader_p$ to p between these two actions. Hence, $ReceiveTimer_p = 0$ holds at a time $t' \in [t - 8\delta, t]$. Now, to set $Leader_p$ to p at time t, p must satisfy $(Leader_p \neq p) \wedge (ReceiveTimer_p > 8\lceil \delta/\alpha \rceil)$ by Observation 5. As $ReceiveTimer_p$ is incremented only once at each iteration of the *repeat forever* loop, $ReceiveTimer_p$ will be greater than $8\lceil \delta/\alpha \rceil$ after at least $8\lceil \delta/\alpha \rceil + 1$ iterations from t'. As each iteration is executed in at least α times, $ReceiveTimer_p$ will be greater than $8\lceil \delta/\alpha \rceil$ after at least $8\delta + \alpha$ times from t'. As $t' + 8\delta + \alpha > t$, we can conclude that p cannot set $Leader_p$ to p during $[t - 8\delta, t]$ — a contradiction. □

Lemma 11 *Starting from any configuration, $\forall t > (n+4)\delta + (2n + 8\lceil \delta/\alpha \rceil + 11)\beta + 1$, every alive process p satisfies: if $Leader_p \neq p$ at time t, then $Leader_p \neq p$ forever from time t.*

Proof. By Lemma 9, $\forall t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$, $\exists t' \in [t, t + 2\delta + 3\beta]$ such that an alive process sends ALIVE to p at time t'. As all the links are timely, such a message is delivered at most δ times after its sending. Also, each alive process receives a message *m* at most one complete iteration of its *repeat forever* loop after the delivrance of *m*, *i.e.*, at most 2β times after the delivrance of *m* by Observation 1, page 13. Thus, $\forall t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1$, p receives ALIVE after at most $3\delta + 5\beta$ times from t. As $3\delta + 5\beta < 8\delta$, we have: $\forall t > (n+1)\delta + (2n + 8\lceil \delta/\alpha \rceil + 6)\beta + 1 + 3\delta + 5\beta$, *i.e.*, $\forall t > (n+4)\delta + (2n + 8\lceil \delta/\alpha \rceil + 11)\beta + 1$, any alive process p do not set $Leader_p$ to p at time t by Lemma 10 and the lemma is proven. □

Lemma 12 *Starting from any configuration, $\forall t > (n+6)\delta + (2n + 8\lceil \delta/\alpha \rceil + 14)\beta + 1$, every alive process p satisfies $Leader_p = l$ forever where l is an alive process.*

Proof. $\forall t > (n+4)\delta + (2n + 8\lceil \delta/\alpha \rceil + 11)\beta + 1$, $Candidates(t) > 0$ by Lemma 7 and, $\forall t' \geq t$, $Candidates(t') \subseteq Candidates(t)$ by Lemma 11. So, there is some processes p such that $Leader_p = p$ at any time $t' \in [t,\ t + \delta + \beta]$. Let $Finalists(t)$ be the set of these processes. Let $l = min(Finalists(t))$. By Lemma 8, l sends at least one ALIVE message to every other alive process during this time interval. These ALIVE messages are delivered at most δ times after their sending because all the links of the system are timely. Finally, each alive process receives a message *m* at most one complete iteration of its *repeat forever* loop after the delivrance of *m*, *i.e.*, at most 2β times after the delivrance of *m* by Observation 1, page 13. Hence, at most $2\delta + 3\beta$ times from t, every alive process p such that $p \neq l$ receives ALIVE from l and sets $Leader_p$ to l is the same loop iteration (Lines 6-13). At the end of the loop iteration, *i.e.*, at most $2\delta + 4\beta$ times from t, every alive process p satisfies $Leader_p = l$ and l is now the only process able to send ALIVE (Lines 16-18). Hence, every alive process p satisfies $Leader_p = l$ forever at most $2\delta + 3\beta$ times from t. □

Proof of Theorem 3. By Lemma 12, starting from any configuration, the system reaches in a *bounded* time a configuration γ from which there is a unique leader forever. As the time to reach γ is bounded, this means that, starting from any configuration, after a bounded time, the system is in a configuration from which it cannot deviate from its specification whatever the execution we consider. Hence, Algorithm 1 is a self-stabilizing leader election algorithm. Also, in Algorithm 1 only a process p such that $Leader_p = p$ can send messages. So, since the system is stabilized, only one process (actually, the leader) sends messages: Algorithm 1 is communication-efficient. □

B Proof of Theorem 5

Lemma 13 *Starting from any configuration, any alive process eventually no more receives (ALIVE,−,q) messages where q is any crashed process.*

Proof. Let q be any process that is crashed in the initial configuration. First, as q is crashed, the messages containing (ALIVE,1,q) are no more sent. Then, each time a process receives an (ALIVE,k,q) message, it sends (ALIVE,$k+1$,q) only if $k \leq n-1$ (Lines 11-16 of Algorithm 2). Finally, every message in transit is eventually received or lost. So, the number of (ALIVE,−,q) messages in the system decreases infinitely often until reaching zero. □

Lemma 14 *Starting from any configuration, any alive process p sends (ALIVE,1,p) to all other processes at least every $\delta + \beta$ times.*

Proof. Consider any time t. From t, the program counter of p points out to the first instruction of the *repeat forever* loop at a time $t' \leq t + \beta$. From t', p executes a complete iteration of the loop at most every β times. Now, from t', while $SendTimer_p < \lfloor \delta/\beta \rfloor$, $SendTimer_p$ is incremented at each loop iteration. So, as $SendTimer_p$ is always greater or equal to 0, the test $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ becomes true at the lattest during the $\lfloor \delta/\beta \rfloor^{th}$ loop iteration from t' and p sends (ALIVE,1,p) to all other processes in the same loop iteration (Lines 19-22). Hence, from t', p sends (ALIVE,1,p) to all other processes in at most $\lfloor \delta/\beta \rfloor \times \beta$ times, *i.e.*, in at most δ times. As $t' \leq t + \beta$, the lemma is proven. □

Definition 4 *Let G' = (V',E') be the strongly connected graph representing the timely routing overlay of the system.*

Lemma 15 *Let p and q be two alive processes such that $p \neq q$. Starting from any configuration, p receives an (ALIVE,d,q) message at least every $(d+1)\delta + 3d\beta$ times where d is the distance from q to p in G'.*

Proof. Let p and q be two alive processes. We prove this lemma by induction on the distance d from q to p in G'.

Induction for $d = 1$: Assume that the distance from q to p is equal to 1 in G'. This means that the link (q,p) exists in G', *i.e.*, there exists a directed timely link from q to p in the communication graph of the system.

1. By Lemma 14, q sends (ALIVE,1,q) to each other process (in particular p) every $\delta + \beta$ times.
2. Each (ALIVE,1,q) message sent from q to p is delivered to p at most δ times after its sending thanks to the timeliness the the link from q to p.
3. p receives a message sent from q at most one complete iteration of the *repeat forever* loop after its delivrance, *i.e.*, at most 2β times after its delivrance by Observation 1.

Hence, p receives an (ALIVE,1,q) message at most every $2\delta + 3\beta$ times and the induction is verified for the distance 1.

Induction Assumption: Let k such that $1 \leq k < D$ where D is the diameter of G'. Assume that every alive process at distance k from q in G' receives an (ALIVE,k,q) message at least every $(k+1)\delta + 3k\beta$ times.

Induction for $d = k + 1$: Let i be process at distance $k+1$ from q. Let j by a neighbor of i at distance k from q.

1. j receives an (ALIVE,k,q) message at least every $(k+1)\delta + 3k\beta$ times by induction assumption.

2. As $k < D$ and $D \leq n-1$, we have $k < n-1$, so, after each reception of any (ALIVE,k,q) message, j sends (ALIVE,$k+1$,q) to i in the same *repeat forever* loop iteration (Lines 11-16), *i.e.*, j sends (ALIVE,$k+1$,q) to i within β times after each reception of (ALIVE,k,q).

3. Each (ALIVE,$k+1$,q) message sent from j to i is delivered to i at most δ times after its sending thanks to the timeliness the link from j to i.

4. i receives a message sent from j at most one complete iteration of the *repeat forever* loop after its delivrance, *i.e.*, at most 2β times after its delivrance by Observation 1.

Hence, i receives an (ALIVE,$k+1$,q) message at least every $(k+1)\delta + 3k\beta + \beta + \delta + 2\beta$ times *i.e.*, every $3(k+2)\delta + 3(k+1)\beta$ times and the induction holds for the distance $k+1$. □

The distance from each alive process to another alive process is bounded by $n-1$ in G'. Hence:

Corollary 3 *Let p and q be two alive processes such that $p \neq q$. Starting from any configuration, p receives an (ALIVE,$-$,q) message at least every $n\delta + 3(n-1)\beta$ times.*

Lemma 16 *Let p be an alive process. Starting from any configuration, $Alives_p$ is eventually equal to the set of all alive processes forever.*

Proof.

1. We first show that $Alives_p$ eventually only contains IDs of alive processes.

   Assume, by the contradiction, that $q \in Alives_p$ holds infinitely often while q is crashed. As p is alive, $p \neq q$ and $q \in OtherAlives_p$ holds infinitely often ($Alives_p = OtherAlives_p \cup \{p\}$). Now, $OtherAlives_p$ is periodically set to $Collect_p$ (Line 25) and $Collect_p$ is periodically reset to $\emptyset$ (Line 27). So, q is inserted into $Collect_p$ infinitely often and, to that goal, p receives (ALIVE,$-$,q) messages infinitely often — a contradiction by Lemma 13.

2. We now show that $Alives_p$ eventually contains the IDs of any alive process forever.

   Let q be an alive processes. First, if $p = q$, then the claim trivially holds. Consider now the case where $p \neq q$. To show the claim, we prove that $q \in OtherAlives_p$ eventually holds forever. From Lines 23-29, we know that p periodically resets $Collect_p$ to $\emptyset$. After p resets $Collect_p$ (Line 27), p resets $ReceiveTimer_p$ to 0 (Line 28), and then waits at least $(4n-3)\lceil \delta/\alpha \rceil + 1$ iterations of its *repeat forever* loop before executing $OtherAlives_p \leftarrow Collect_p$ (Line 25). As p executes every iteration of its *repeat forever* loop in at least α times, p waits at least $(4n-3)\delta + \alpha$ times before executing $OtherAlives_p \leftarrow Collect_p$. During this period, p receives at least one (ALIVE,$-$,q) message for any other alive process q by Corollary 3. So, during this period, p inserts each alive process $q \neq p$ in $Collect_p$ (Line 12). Hence, since the first execution of $OtherAlives_p \leftarrow Collect_p$ after the first execution of $Collect_p \leftarrow \emptyset$, $OtherAlives_p$ contains the IDs of any alive process forever.

□

Proof of Theorem 5. In Algorithm 2, each alive process p periodically sets $Leader_p$ to min($Alives_p$) (Lines 23-29). Hence, by Lemma 16, each alive process eventually designates the alive process with the smallest ID as its own leader. As each process that is alive in the initial configuration is alive forever, this process is the same during the whole execution. So, if l is the alive process with the smallest ID in an arbitrary configuration γ, then, in any execution starting from γ, every alive process p eventually satisfies $Leader_p = l$ forever and the theorem holds. □

C Proof of Theorem 6

In the following, we denote by var_p^t the value of var_p at time t. We also denote by b the timely bi-source of the system.

Definition 5 *We say that a process p* starts *Round k at time t if p executes $StartRound(k)$ at time t. We say that a process p* is in *Round k at time t if $Round_p = k$ at time t. We say that a process p* times out *on Round k at time t if $Round_p = k \wedge ReceiveTimer_p > 8\lceil \delta/\alpha \rceil$ when p executes Line 27 at time t.*

Lemma 17 *Starting from any configuration, $\exists k \in \mathbb{N}$ such that: if some process starts a round greater than k, then some process previously times out on round k.*

Proof. Starting from any configuration, since all alive processes have begun their 2^{nd} *repeat forever* loop iteration, we are sure that an alive process executes Line 29 only after it times out. So, let t be the first time after which all alive processes have begun their 2^{nd} *repeat forever* loop iteration. Let k be the maximal round value in the network (considering messages and processes). Any round value $k' > k$ appears in the network only when at least one process times out on $k' - 1$. The lemma is then proven through a simple induction argument. □

Corollary 4 *Starting from any configuration, $\exists k \in \mathbb{N}$ such that $\forall k' \geq k$, if a process starts Round $k' + 1$, then some process previously started Round k'.*

Lemma 18 *Starting from any configuration, if an alive process p continuously satisfies $Round_p \bmod n + 1 = p$ during the period $[t, t + \delta + \beta]$, then p sends at least one (ALIVE,$Round_p^t$) message to any other process during this period.*

Proof. Similar to the proof of Lemma 8, page 15. □

Lemma 19 *Processes start finitely many rounds.*

Proof. Assume, by the contradiction, that some process p starts infinitely many rounds. Then, by Lemma 17 and Corollary 4, $\exists k \in \mathbb{N}$ such that $\forall k' \geq k$, some process starts Round k' and some process times out on Round k'.

Consider the time t_0 where the round value k appears in the system. Consider now any time t_1 such that $t_1 \geq t_0$. Let L be the largest value sent by time t_1 in any message. Let L' be the first value greater than L such that $L' \bmod n = b$. Let t_2 be the earliest time when some process p times out on Round $L' - 1$. By Lemma 17, (1) a process can only start Round L' after time t_2. Now, $t_2 > t_1$ by definition of L', and thus process p is alive, so it not only times out on Round $L' - 1$ but it also starts Round L' and two cases are possible:

1. $p = b$. Then, p sends (ALIVE,L') to all other processes before time $t_2 + \beta$ (before the end of the loop iteration).
2. $p \neq b$. In this case, p sends (START,L') to b before time $t_2 + \beta$ (before the end of the current loop iteration). This message is delivered to b at most δ times later. So, b receives such a message at most $\delta + 2\beta$ times later by Observation 1 (page 13), *i.e.*, at most at time $t_2 + \delta + 3\beta$. Finally, during the loop iteration where it receives (START,L'), *i.e.* during $]t_2, t_2 + \delta + 4\beta]$, b starts Round L' and sends (ALIVE,L') to all other processes.

Hence, in the worst case, (2) any alive process different of b is guaranteed to receive the first (ALIVE,L') by time $t_2 + 2\delta + 6\beta$ ($t_2 + \delta + 4\beta$ plus δ times for the delivrance and 2β times for the reception after the delivrance) and, henceforth, another such a message at least every $2\delta + 3\beta$ times while L' has not been timed out on (by Lemma 18, while L' has not been timed out on, b sends (ALIVE,L') every $\delta + \beta$ times and, similary to the previous cases, such a message is received $\delta + 2\beta$ times after its sending). To time out on Round L', a process must have started L' and must failed to receive a message from b for more than $8\lceil \delta/\alpha \rceil$ complete loop iterations, *i.e.*, for more than 8δ times. Therefore, through a simple induction argument, (1) and (2) implies that no process ever times out on L'. This contradicts the fact that every round is started and timed out. □

Let K be the largest round started by any alive process and let $P = K \bmod n$.

Lemma 20 *P sends an infinite number of (ALIVE,K) messages to all others alive processes.*

Proof. Let p an alive process that is in Round K. If P only sends a finite number of (ALIVE,K) messages to p, then p eventually starts a round larger than K — a contradiction. □

Lemma 21 *There is a time after which, for every alive process p, $Leader_p = P$.*

Proof. Immediate from the definition of K and Lemma 20. □

Corollary 5 *There is a time after which only P sends messages.*

Proof of Theorem 6. Immediate from Lemma 21 and Corollary 5. □

D Proof of Theorem 8

In the following, we denote by var_p^t the value of var_p at time t. Also, we denote by s the timely source of the system.

Lemma 22 *Starting from any configuration, for every alive process p and every process q such that $q \neq p$: if $q \in Actives_p$ holds infinitely often, then p receives ALIVE messages from q infinitely often.*

Proof. Let p and q be two processes such that p is alive and $q \neq p$. Assume that $q \in Actives_p$ holds infinitely often. As $q \neq p$, $q \in OtherAlives_p$ also holds infinitely often (Line 8). As $OtherActives_p$ is periodically reset to $Collect_p$ (Line 32), $q \in Collect_p$ holds infinitely often. Now, $Collect_p$ is periodically reset to $\emptyset$ (Line 41). So, q is inserted into $Collect_p$ infinitely often. To that goal, p must receive ALIVE message from q infinitely often (Lines 12-16). □

Observation 6 *For every process p, $Counter_p[p]$ and $Phase_p[p]$ are monotonically nondecreasing with time.*

Lemma 23 *Let p and q be two distinct processes. Starting from any configuration, if p receives ALIVE messages from q infinitely often, then q is alive and, for every time t, there is a time after which $Counter_p[q] \geq Counter_q^t[q]$ and $Phase_p[q] \geq Phase_q^t[q]$ forever.*

Proof. Let p and q be two processes such that $p \neq q$. Assume that p receives ALIVE messages from q infinitely often. As the number of messages initially in the link (q,p) is finite, p eventually only receives messages that have been sent by q. So, q sends such messages infinitely often and, as a consequence, q is alive. Consider now any time t. As every message in the link (q,p) is eventually received or lost, there is a time $t' > t$ from which p only receives from q ALIVE messages that have been sent by q after time t. Now, any (ALIVE,$qcnt$,qph) message sent by q to p after time t satisfies $qcnt \geq Counter_q^t[q]$ and $qph \geq Phase_q^t[q]$ because $Counter_q[q]$ and $Phase_p[p]$ are monotonically nondecreasing (Observation 6). Thus, from t', p only receives from q (ALIVE,v,w) messages such that $v \geq Counter_q^t[q]$ and $w \geq Phase_q^t[q]$. Now, each time p receives such an (ALIVE,v,w) message from q, $Counter_p[q]$ is set to v and $Phase_p[q]$ is set to w (Lines 12-16) and this is the only way that p can modify $Counter_p[q]$ or $Phase_p[q]$. Hence, $Counter_p[q] \geq Counter_q^t[q]$ and $Phase_p[q] \geq Phase_q^t[q]$ eventually hold forever. □

Lemma 24 *Starting from any configuration, for every alive process p and every process q, if $q \in Actives_p$ holds infinitely often, then q is alive and, for every time t, there is a time after which $Counter_p[q] \geq Counter_q^t[q]$ and $Phase_p[q] \geq Phase_q^t[q]$ forever.*

Proof. Assume that $p = q$. In this case, the lemma holds because p is alive and $Counter_p[p]$ and $Phase_p[p]$ are monotically nondecreasing by Observation 6. Assume now that $p \neq q$. If $q \in Actives_p$ holds infinitely often, then by Lemma 22, p receives ALIVE messages from q infinitely often and the lemma holds by Lemma 23. □

Lemma 25 *For every alive process p and q, if p sends a message of type T to q infinitely often, then q receives a message of type T from q infinitely often.*

Proof. Since the link (p,q) is *fair*, the lemma is trivial. □

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), we are sure that any process p sends ALIVE (Line 26) only if the test of Line 25 is true, *i.e.*, only if $Leader_p = p$. Hence:

Observation 7 *Starting from any configuration, a process p sends ALIVE at a time $t > \beta$ only if $Leader_p = p$ at time t.*

Starting from any configuration, since the second iteration of the *repeat forever* loop begins (after at most β times), we are sure that any process executes Line 37 only if it previously executes Line 36. Hence, follows:

Observation 8 *Starting from any configuration, any process p switches $Leader_p$ from p to $q \neq p$ at a time $t > \beta$ only if $OldLeader_p = p$ at time t.*

Lemma 26 *For every process $p \neq s$ and every $k \geq 0$, if s sends (ALIVE,$-$,k) to p at some time $t > \beta$, then:*

- *s sends another (ALIVE,$-$,k) message to p during time interval $]t, t + \delta + \beta]$, or*
- *$Phase_s[s] > k$ holds at time $t + \delta + \beta$.*

Proof. First, s satisfies $Leader_s = s$ at time t by Observation 7. Then, $k = Phase_s[s]^t$ (Line 26). Consider now the two following cases:

- Assume that s switches $Leader_s$ from s to $q \neq s$ at a time $t' \in]t,t+\delta]$ (Line 37). Then, s satisfies $OldLeader_s = s$ at time t' by Observation 8 and, so, increments $Phase_s[s]$ (Line 39) before the end of the current *repeat forever* loop iteration, *i.e.*, before time $t' + \beta$. Now, as $t' \in]t,t+\delta]$ and $Phase_s[s]$ is monotically nondecreasing (Observation 6), the lemma holds in this case.

- Assume that s continuously satisfies $Leader_s = s$ during the time interval $]t,t+\delta]$. Then, as s sends (ALIVE,$-$,k) to p at time t (Line 26), s resets $SendTimer_s$ to 0 (Line 28) before the beginning of the next *repeat forever* loop iteration. So, when the program counter of s points out to the first instruction of the *repeat forever* loop at a time t' such that $t < t' \leq t + \beta$, $SendTimer_s = 0$. From t', s executes a complete loop iteration at most every β times. So, after executing $\lfloor \delta/\beta \rfloor - 1$ complete iterations, s points out to the first intruction of the loop at a time $t'' \leq t + \delta$, $SendTimer_s = \lfloor \delta/\beta \rfloor - 1$ ($SendTimer_s$ is incremented at each loop iteration), and s can still execute a complete iteration of the loop in the time interval $[t'',t+\delta+\beta]$. During this loop iteration, s increments $SendTimer_s$ to $\lfloor \delta/\beta \rfloor$ (Line 23) and, as s satisfies the test of Lines 24 and 25, s sends another alive message to p (Line 26) before the end of the iteration, *i.e.*, before time $t + \delta + \beta$. As s points out to Line 26 at time t (s sends ALIVE to p at time t) and s continuously satisfies $Leader_s = s$ during the time interval $[t,t+\delta]$, s does not increments $Phase_s[s]$ during $]t,t+\delta+\beta]$. So, when s sends another ALIVE message to p during time interval $]t,t+\delta+\beta]$, $Phase_s[s] = Phase_s[s]^t$ and, as a consequence, the message is of the following form: (ALIVE,$-$,k) and the lemma also holds in this case.

□

As all the output links of s are timely, we can deduce the following:

Observation 9 *If s sends a message m to another process p at some time t, then m is delivered to p from s at most at time $t + \delta$.*

Assume that a message *m* is delivered to a process p. Then, p receives a message of the same type of *m* at most one complete iteration of its *repeat forever* loop after the delivrance of *m*. Hence, by Observations 1 (page 13) and 9:

Lemma 27 *Starting from any configuration, if s sends ALIVE to another process p at time t, then p receives at least one ALIVE message from s during the time interval $]t, t + \delta + 2\beta]$.*

Lemma 28 *$Counter_s[s]$ is bounded.*

Proof. Assume, by the contradiction, that $Counter_s[s]$ is unbounded. Then, s executes Line 19 of the algorithm infinitely often. From Lines 17-18, we can then deduce that the following situation appears infinitely often: s receives an (ACCUSATION,ph) message from a process p at some time t with $ph = Phase_s[s]^t$. As the number message initially in the link (p,s) is finite, we can then deduce that p sends such messages infinitely often.

p sends ACCUSATION messages to s infinitely often only if $Leader_p = s \wedge Leader_p \notin Actives_p$ holds infinitely often. Now, $Leader_p$ is periodically set to a process in $Actives_p$ (Line 37). So, (1) s is inserted in $Actives_p$, (2) $Leader_p$ is set to s, and (3) s removed from $Actives_p$ infinitely often. By (1), Lemma 22, and the fact that the number of messages initially in the link (s,p) is finite, we can deduce that p receives infinitely often ALIVE messages sent by s.

p updates $Actives_p$ by setting $OtherActives_p$ to $Collect_p$ (Line 32). After each $Actives_p$'s update (Line 32):

- p sends an ACCUSATION message to s (Line 34) if $Leader_p = s \wedge Leader_p \notin Actives_p$ (Line 33-35),

- p chooses a leader in $Actives_p$ (Line 37), and

- p resets $Collect_p$ to $\emptyset$, and $ReceiveTimer_p$ to 0 (Lines 41-42).

Then, p waits at least $5\lceil \delta/\alpha \rceil$ complete loop iterations, *i.e.*, at least 5δ times to make the next $Actives_p$'s update.

Consider now the time t from which p only receives from s ALIVE messages that was effectively sent by s (such a time exists because each message in transit in the link (s, p) is eventually received or lost). From time t, s is inserted into $Collect_p$ each time p receives an ALIVE message sent by s. As p receives an ALIVE message sent by s infinitely often, p sends ACCUSATION messages to s only if the following situation appears infinitely often: p receives an ALIVE message sent by s and, then, receives no ALIVE message from s during at least 5δ times. By Lemma 26, two cases are then possible for each (ALIVE,$-$,k) message sent by s to p at time $t' \geq t$:

(a) s sends another (ALIVE,$-$,k) message to p during time interval $]t',t' + \delta + \beta]$.

(b) $Phase_s[s] > k$ holds at time $t' + \delta + \beta$.

Let us now study the two following cases:

- There is a time $t_a \geq t$ from which Case (a) is always verified, *i.e.*, from t_a, s sends (ALIVE,$-$,k) to p at most every $\delta + \beta$ times. Then, by Lemma 27, we can conclude that p receives an ALIVE message from s at least every $2\delta + 3\beta$ times. So, s is eventually inserted into $Collect_p$ at least once during each period of 5δ times and, as a consequence, $s \in Actives_p$ eventually holds forever. Thus, we can conclude that p eventually no more sends any accusation to s and, so, $Counter_s[s]$ is eventually no more incremented — a contradiction.

- Case (b) is verified infinitely often. Then, from time t, p must receive an ALIVE message sent by s and, then, receive no message during at least 5δ in order to send an ACCUSATION message to s. Consider any time $t_r \geq t$. Assume that (i) p receives at time t_r a message m = (ALIVE,$-$,k) sent by s at time $t_s < t_r$ and (ii) p does not receive any ALIVE message from s during the time interval $]t_r$,$t_r + 5\delta]$. Then, by Lemma 27, if s sends another (ALIVE,$-$,k) message during $]t_s$,$t_s + \delta + \beta]$, p receives the message before time $t_r + 5\delta$ — a contradiction. So, Case (b) is verified and, as $Phase_s[s]$ is monotically nondecreasing, $Phase_s[s] > k$ holds forever from time $t_s + \delta + \beta$. After receiving m, $Phase_p[s]$ is set to k. So, the ACCUSATION message m_A provoked by m is of the following form: (ACCUSATION,k). Now, as m_A is sent after time $t_r + 5\delta$, $Phase_s[s] > k$ holds when s receives m_A and, as a consequence, m_A does not provoke any incrementation of $Counter_s[s]$. Thus, as we consider t_r as any value greater or equal to t, this means that eventually no ACCUSATION message received from p can provoke any incrementation of $Counter_s[s]$ — a contradiction.

Hence, in any case, $Counter_s[s]$ is incremented only a finite number of times — a contradiction. □

Definition 6 *For each process p, let c_p be the largest value of $Counter_p[p]$ in the execution that we consider ($c_p = \infty$ if $Counter_p[p]$ is unbounded). Let l be the process such that (c_l,l) = min$\{(c_p,p)$: p is an alive process$\}$.*

By Definition, l is an alive process. Furthermore, by Lemma 41 $c_s < \infty$, so, $c_l < \infty$, *i.e.*, $Counter_l[l]$ is bounded.

Lemma 29 *Starting from any configuration, for every alive process p, if there is a time after which $l \in Actives_p$ forever, then there is a time after which $Leader_p = l$ forever.*

Proof. Assume, by the contradiction, there is an alive process p that satisfies $Leader_p \neq l$ infinitely often despite $l \in Actives_p$ eventually holds forever. Then, as $Leader_p$ is periodically set to a process in $Actives_p$ (Line 37), this means that there is a process $q \neq l$ such that $q \in Actives_p$ and $Leader_p = q$ infinitely often. $l \in Actives_p$ eventually holds forever implies that p receives ALIVE messages from l infinitely often. As the number of ALIVE messages initially in the link (l,p) is finite, p eventually only receives from l ALIVE messages that l effectively sends, also, as $Counter_l[l]$ is bounded and monotically nondecreasing (Observation 6), p eventually only receives ALIVE messages from l of the form (ALIVE,c_l) and, as a consequence, $Counter_p[l] = c_l$ eventually holds forever. Consider now the two following cases:

1. $Counter_q[q]$ is bounded. In this case, $c_q < \infty$ and, so, there is a time t when $Counter_q^t[q] = c_q$. By Lemma 24, there is a time after which $Counter_p[q] \geq Counter_q^t[q]$ forever, *i.e.*, $Counter_p[q] \geq c_q$ eventually holds forever. Now, by definition of l, we have (c_l,l) < (c_q,q). So, there is a time which ($Counter_p[l]$,l) < ($Counter_p[q]$,q) forever, and from the way that p periodically sets $Leader_p$ (Line 37) — we obtain a contradiction.

2. $Counter_q[q]$ is unbounded. Then, by Lemma 24, $Counter_p[q]$ is also unbounded. So, there is a time which ($Counter_p[l]$,l) < ($Counter_p[q]$,q) forever — we also obtain a contradition.

□

Lemma 30 *Starting from any configuration, there is a time after which $Leader_l = l$ forever.*

Proof. By definition, $l \in Actives_l$ (Line 8). So, the result follows from Lemma 29. □

Corollary 6 *Starting from any configuration, there is a time after which $Phase_l[l]$ stops changing.*

Proof. l changes $Phase_l[l]$ infinitely often only if l switches $Leader_l$ from l to a process $q \neq l$ infinitely often (Lines 36-40). Hence, the result immediatly holds from Lemma 30. □

Definition 7 *Let $lphase$ be the final value of $Phase_l[l]$.*

Note that since $Phase_l[l]$ is monotically nondecreasing, $lphase$ is also the largest value of $Phase_l[l]$.

Lemma 31 *Starting from any configuration, for every alive process p, there is a time after which $l \in Actives_p$ forever.*

Proof. Let p be any alive process. If $p = l$, then the lemma is trivially verified. Assume now that $p \neq l$. By Lemma 30 and the definition of $lphase$, l sends (ALIVE,$-$,$lphase$) messages to p infinitely often and these are the only type of ALIVE message that l sends to p infinitely often. By Lemma 25, p receives (ALIVE,$-$,$lphase$) from l infinitely often. Therefore, (*) there is a time after which $Phase_p[l] = lphase$ holds forever. Moreover, p adds l to $Actives_p$ infinitely often. We now show that p removes l from $Actives_p$ only finitely often, and so the lemma holds. To that goal, assume, by the contradiction, that p removes l from $Actives_p$ infinitely often. Then, p sends (ACCUSATION,$-$) messages to l infinitely often. By Lemma 25, l receives (ACCUSATION,$-$) messages from p infinitely often. By (*), there is a time after which the only (ACCUSATION,$-$) messages that p sends to l are of the form (ACCUSATION,$lphase$). Thus, l receives (ACCUSATION,$lphase$) messages from p infinitely often and $Counter_l[l]$ is unbounded — a contradiction. □

By Lemmas 29 and 31, we have:

Lemma 32 *Starting from any configuration, for every alive process p, there is a time after which $Leader_p = l$ forever.*

Lemma 33 *Starting from any configuration, there is a time after which only l sends messages.*

Proof. There are only two types of messages in Algorithm 4: ALIVE and ACCUSATION. By Lemmas 31 and 32, the test of Line 33 is eventually no more satisfied by any alive process. As a consequence, there is a time after which no ACCUSATION message are sent. Consider now the ALIVE messages. From Line 25 of the algorithm, we know that only the alive processes p that satisfy $Leader_p = p$ infinitely often can send ALIVE messages infinitely often. By Lemma 32, there is a time after which only one alive process p satisfy $Leader_p = p$ infinitely often: Process l. Hence, eventually only one process, l, sends messages (namely, ALIVE) and the lemma is proven. □

Proof of Theorem 8. Immediate from Lemmas 32 and 33. □

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

Algorithm 5 implements a pseudo-stabilizing but non communication-efficient leader election in any system $\mathcal{S}_1$. Below, its correctness proof. Below, we note var_p^t the value of var_p at time t and s the timely source of the system.

Lemma 34 *Starting from any configuration, for every alive process p and every process q such that $q \neq p$: if $q \in Alives_p$ holds infinitely often, then p receives ALIVE messages from q infinitely often.*

Proof. Similar to the proof of Lemma 22, page 19. □

Observation 10 *For every process p, $Counter_p[p]$ is monotonically nondecreasing with time.*

Lemma 35 *Let p and q be two distinct processes. Starting from any configuration, if p receives ALIVE messages from q infinitely often, then q is alive and, for every time t, there is a time after which $Counter_p[q] \geq Counter_q^t[q]$ forever.*

Proof. Similar to the proof of Lemma 23, page 19. □

Lemma 36 *Starting from any configuration, for every alive process p and every process q, if $q \in Alives_p$ holds infinitely often, then q is alive and, for every time t, there is a time after which $Counter_p[q] \geq Counter_q^t[q]$ forever.*

Proof. Similar to the proof of Lemma 24, page 19. □

Lemma 37 *Starting from any configuration, if s (the source) sends ALIVE to another process p at time t, then s sends another ALIVE message to p during the time interval $]t, t + \delta + \beta]$.*

Proof. Assume that s sends ALIVE to another process p at time t. Just after sending ALIVE to p (Line 26), s resets its timer $SendTimer_s$ to 0 (Line 27) in the same *repeat forever* loop iteration. The program counter of s then points out to the first instruction of the loop at a time t' such that $t < t' \leq t + \beta$. From t', s then executes a complete iteration of the loop at most every β times. Now, from t', while $SendTimer_s < \lfloor \delta/\beta \rfloor$, $SendTimer_s$ is incremented at each loop iteration. So, the test $SendTimer_s \geq \lfloor \delta/\beta \rfloor$ becomes true during the $\lfloor \delta/\beta \rfloor^{th}$ loop iteration from t' and, then, s sends ALIVE to p in the same loop iteration (Lines 24-28). Hence, from t', s sends ALIVE to p in at most $\lfloor \delta/\beta \rfloor \times \beta$ times, *i.e.*, in at most δ times. As $t' \leq t + \beta$, the lemma is proven. □

As all the output links of s are timely, we can deduce the following:

Algorithm 5 Pseudo-Stabilizing Leader Election on $\mathcal{S}_1$

CODE FOR EACH PROCESS p:

```
 1: variables:
 2:     Leader_p ∈ {1,...,n}
 3:     SendTimer_p, ReceiveTimer_p: non-negative integers
 4:     LocalLeader_p[1...n], LocalLeaderCounter_p[1...n], Counter_p[1...n]: arrays of non-negative integers   /* to manage the accusations */
 5:     Collect_p, OtherAlives_p: sets of non-negative integers   /* these sets are used to compute the Actives_p set */
 6:
 7: macros:   /* these macros are just used to simplify the code */
 8:     Alives_p = OtherAlives_p ∪ {p}
 9:     MyLocalLeader_p = r such that (Counter_p[r],r) = min{(Counter_p[q],q) : q ∈ Alives_p}
10:     MyLeader_p = l such that (LocalLeaderCounter_p[l],LocalLeader[l]) = min{(LocalLeaderCounter_p[q],LocalLeader[q]) : q ∈ Alives_p}
11:
12: repeat forever
13:     for all q ∈ V \ {p} do
14:         if receive(ACCUSATION) from q then   /* each time p receives an ACCUSATION, p increments its accusation counter */
15:             Counter_p[p] ← Counter_p[p] + 1
16:         end if
17:         if receive(ALIVE,r,rcnt,qcnt) from q then   /* we also use the ALIVE messages to carry some informations */
18:             Collect_p ← Collect_p ∪ {q}
19:             Counter_p[q] ← qcnt
20:             LocalLeader_p[q] ← r
21:             LocalLeaderCounter_p[q] ← rcnt
22:         end if
23:     end for
24:     SendTimer_p ← SendTimer_p + 1
25:     if SendTimer_p ≥ ⌊δ/β⌋ then   /* p periodically sends ALIVE to each other */
26:         send(ALIVE,LocalLeader_p[p],Counter_p[LocalLeader_p[p]],Counter_p[p]) to every process except p
27:         SendTimer_p ← 0
28:     end if
29:     ReceiveTimer_p ← ReceiveTimer_p + 1
30:     if ReceiveTimer_p > 5⌈δ/α⌉ then
31:         OtherAlives_p ← Collect_p
32:         for all q ∈ V \ Alives_p do   /* p periodically accuses the processes it suspects */
33:             send(ACCUSATION) to q
34:         end for
35:         LocalLeader_p[p] ← MyLocalLeader_p   /* p periodically evaluates its local leader */
36:         LocalLeaderCounter_p[p] ← Counter_p[LocalLeader_p[p]]
37:         Leader_p ← MyLeader_p   /* p periodically evaluates its global leader */
38:         Collect_p ← ∅
39:         ReceiveTimer_p ← 0
40:     end if
41: end repeat
```

Observation 11 *If s sends m to a process $p \neq s$ at time t, then m is delivered to p from s at most at time $t + \delta$.*

Assume that a message *m* is delivered to a process p. Then, p receives a message of the same type of *m* at most one complete iteration of its *repeat forever* loop after the delivrance of *m*. Hence, by Observations 1 (page 13) and 11, follows:

Lemma 38 *Starting from any configuration, if s sends ALIVE to another process p at time t, then p receives at least one ALIVE message from s during the time interval $]t, t + \delta + 2\beta]$.*

Lemma 39 *Starting from any configuration, for every alive process $p \neq s$, p receives ALIVE messages from s at least every $2\delta + 3\beta$ times.*

Proof. Starting from any configuration, the program counter of s points out to the first instruction of its *repeat forever* loop at a time t such that $t \leq \beta$. From t, s then executes a complete iteration of the loop at most every β times and, while $SendTimer_s < \lfloor \delta/\beta \rfloor$, $SendTimer_s$ is incremented at each loop iteration. So, as $SendTimer_s$ is a non-negative integer, the test $SendTimer_s \geq \lfloor \delta/\beta \rfloor$ becomes true at the lattest during the $\lfloor \delta/\beta \rfloor^{th}$ loop iteration from t and, then, s sends ALIVE to p in the same loop iteration (Lines 24-28). Hence, from the initial configuration, s sends ALIVE to p at most at time $t + \lfloor \delta/\beta \rfloor \times \beta$, *i.e.*, at most at time $\delta + \beta$. After this sending, s periodically sends ALIVE messages to p within periods of at most $\delta + \beta$ times, by Lemma 37. Hence, starting from any configuration, s sends ALIVE messages to p at most every $\delta + \beta$ times and, by Lemma 38, the lemma holds. □

Lemma 40 *For every alive process p, there is a time after which $s \in Alives_p$ forever.*

Proof. First, the lemma trivially holds for $p = s$. Consider now the case where $p \neq s$. There is a time after which $s \in Alives_p$ forever if and only if there is a time after which $s \in OtherAlives_p$ forever. By Lines 29-40, we know that $OtherAlives_p$ is periodically reset to $Collect_p$ and, after that, $Collect_p$ is reset to $\emptyset$. After such resets, p waits $5\lceil \delta/\alpha \rceil$ complete iterations of its *repeat forever* loop before executing $OtherAlives_p \leftarrow Collect_p$ again. As each loop iteration is executed in at least α times, this means that p waits at least 5δ times before executing $OtherAlives_p \leftarrow Collect_p$

again. During this period, p receives at least one ALIVE message from s by Lemma 39. So, during this period, p inserts s in $Collect_p$ (Lines 17-18). Hence, when p executes $OtherAlives_p \leftarrow Collect_p$ again, $s \in Collect_p$. □

Lemma 41 *$Counter_s[s]$ is bounded.*

Proof. Assume, by the contradiction, that $Counter_s[s]$ increases infinitely often. So, s receives ACCUSATION messages infinitely often (Lines 14-16). As the number of messages initially in the links is finite, there is at least one alive process $p \neq s$ that accuses s infinitely often. Now, p only sends ACCUSATION messages to processes q such that $q \in V \setminus Alives_p$ (Lines 32-34) and $s \in Alives_p$ eventually holds forever by Lemma 40 — a contradiction. □

Definition 8 *For each process p, let c_p be the largest value of $Counter_p[p]$ in the execution that we consider ($c_p = \infty$ if $Counter_p[p]$ is unbounded). Let l be the process such that $(c_l,l) = min\{(c_p,p)$: p is an alive process$\}$.*

By Definition, l is an alive process. Furthermore, by Lemma 41, $c_s < \infty$, so, $c_l < \infty$, *i.e.*, $Counter_l[l]$ is bounded.

Lemma 42 *Let p and q be two alive processes. Starting from any configuration, the two following propositions holds:*

(a) if $q \in Alives_p$ infinitely often and $c_q < \infty$, then there is a time after which $Counter_p[q] = c_q$ forever.

(b) if $q \in Alives_p$ infinitely often and $c_q = \infty$, then there is a time after which $Counter_p[q] > c_l$ forever.

Proof. First, if $p = q$, then (a) holds because $Counter_q[q]$ is monotically nondecreasing by Observation 10. Then, if $p = q$, then (b) holds because $Counter_q[q]$ is monotically nondecreasing and c_l is bounded (by definition).

Consider now the case where $p \neq q$. In the two cases (a) and (b), p receives ALIVE from q infinitely often by Lemma 34.

(a) Assume now that $c_q < \infty$. In this case, $Counter_q[q]$ is bounded and monotically nondecreasing (Observation 10). So, there is a time t after which $Counter_q[q] = c_q$ forever. Then, as every message in the link (q,p) is eventually received or lost, there is a time $t' > t$ after which p only receives from q ALIVE messages that have been sent by q after time t and all these messages are of the following form: (ALIVE,$-$,$-$,c_q). Now, each time p receives such an (ALIVE,$-$,$-$,c_q) message, p sets $Counter_p[q]$ to c_q (Lines 17-22) and, this is the only way that p can update $Counter_p[q]$. Hence, there is a time after which $Counter_p[q] = c_q$ forever.

(b) Assume that $c_q = \infty$. In this case, $Counter_q[q]$ is unbounded. Then, we already know that $Counter_l[l]$ is bounded. So, there is a time after which $Counter_q[q] > Counter_l[l]$ forever (remember that $Counter_q[q]$ and $Counter_l[l]$ are monotically nondecreasing by Observation 10). Therefore, by Lemma 36, there is a time after which $Counter_p[q] \geq Counter_q[q] > Counter_l[l]$ forever. Now, $Counter_l[l]$ is eventually equals to c_l forever because $Counter_l[l]$ is monotically nondecreasing. Hence, there is a time after which $Counter_p[q] > c_l$ forever.

□

As $LocalLeader_p[p]$ is periodically set to a process q such that $q \in Alives_p$, we have the following corollary:

Corollary 7 *Let p and q be two alive processes. Starting from any configuration, the two following propositions holds:*

(a) if $LocalLeader_p[p] = q$ infinitely often and $c_q < \infty$, then there is a time after which $Counter_p[q] = c_q$ forever.

(b) if $LocalLeader_p[p] = q$ infinitely often and $c_q = \infty$, then there is a time after which $Counter_p[q] > c_l$ forever.

Lemma 43 *Let p be an alive process. Let q be a process. Assume that $q \in Alives_p$ and $LocalLeader_p[q] = r$ holds infinitely often. The two following propositions hold:*

(a) There is a time after which ($LocalLeader_p[q] = r$) $\Rightarrow$ ($LocalLeaderCounter_p[q] = c_r$) holds each time p sets $Leader_p$ to $MyLeader_p$, if $c_r < \infty$,.

(b) There is a time after which ($LocalLeader_p[q] = r$) $\Rightarrow$ ($LocalLeaderCounter_p[q] > c_l$) holds each time p sets $Leader_p$ to $MyLeader_p$, if $c_r = \infty$.

Proof. Assume that $q = p$. Then, by Corollary 7, there is a time after which:

- $Counter_p[r] = c_r$ forever, if $c_r < \infty$
- $Counter_p[r] > c_l$ forever, if $c_r = \infty$

So, the lemma holds because p periodically executes the following sequence: p updates $LocalLeader_p[p]$, resets $LocalLeaderCounter_p[p]$ to $Counter_p[LocalLeader_p[p]]$, and then sets $Leader_p$ to $MyLeader_p$ (Lines 35-37).

Consider now the case where $q \neq p$. Then, by Lemmas 34 and 35, p receives ALIVE messages from q infinitely often and q is alive. As the number of messages initially in the link (q,p) is finite, p eventually only receives from q ALIVE messages sent by q. Each ALIVE message sent by q at time t is of the following form: (ALIVE,v,$vcnt$,$qcnt$) where v is the value of $LocalLeader_q[q]$ at time t and $vcnt$ is the value of $Counter_q[LocalLeader_q[q]]$ at time t. When receiving such a message, p sets $LocalLeader_p[q]$ to v and $LocalLeaderCounter_p[q]$ to $vcnt$ in sequel (Lines 20-21). Moreover, this is the only way to modify $LocalLeader_p[q]$ and $LocalLeaderCounter_p[q]$. Thus, $LocalLeader_p[q] = r$ holds infinitely often implies that $LocalLeader_q[q] = r$ holds infinitely often and, by Corollary 7:

- if $c_r < \infty$, then $Counter_q[r] = c_r$ eventually holds forever.
- if $c_r = \infty$, then $Counter_q[r] > c_l$ eventually holds forever.

So, if $c_r < \infty$, then p eventually only receives from q (ALIVE,v,$vcnt$,$qcnt$) messages that satisfy the condition $(v = r) \Rightarrow (vcnt = c_r)$. At each reception of such messages, p sets $LocalLeader_p[q]$ to r and $LocalLeaderCounter_p[q]$ to c_r in sequel. So, eventually each time p sets $Leader_p$ to $MyLeader_p$, we have $LocalLeaderCounter_p[q] = c_r$, if $LocalLeader_p[q] = r$ and Part (a) of the lemma is proven.

Finally, if $c_r = \infty$, then p eventually only receives from q (ALIVE,v,$vcnt$,$qcnt$) messages that satisfy the condition $(v = r) \Rightarrow (vcnt > c_l)$. At each reception of such messages, p sets $LocalLeader_p[q]$ to r and $LocalLeaderCounter_p[q]$ to c_r in sequel. So, eventually each time p sets $Leader_p$ to $MyLeader_p$, we have $LocalLeaderCounter_p[q] > c_l$, if $LocalLeader_p[q] = r$ and Part (b) of the lemma is proven. □

Lemma 44 *Starting from any configuration, for every alive process p, if there is a time after which $l \in Alives_p$ forever, then there is a time after which $LocalLeader_p[p] = l$ forever.*

Proof. Let p be any alive process. Assume, by the contradiction, that there is a time after which $l \in Alives_p$ forever but $LocalLeader_p[p] \neq l$ holds infinitely often. Then, by Lemma 42, there is a time after which $Counter_p[l] = c_l$ forever ($c_l < \infty$). Also, there is a process q such that $LocalLeader_p[p] = q$ infinitely often and two cases are possible:

(1) $c_q < \infty$. In this case, there is a time after which $Counter_p[q] = c_q$ forever by Corollary 7. Now, as $Counter_p[l] = c_l$ eventually holds forever, there is a time after which ($Counter_p[l]$,l) < ($Counter_p[q]$,q) forever. Hence, there is a time after which $LocalLeader_p[p] \neq q$ forever — a contradiction.

(2) $c_q = \infty$. In this case, there is a time after which $Counter_p[q] > c_l$ forever by Corollary 7. Now, as $Counter_p[l] = c_l$ eventually holds forever, there is a time after which ($Counter_p[l]$,l) < ($Counter_p[q]$,q) forever. Hence, there is a time after which $LocalLeader_p[p] \neq q$ forever — a contradiction.

□

Definition 9 *Let $LocalLeaders(p) = \{LocalLeader_p[q] : q \in Alives_p\}$.*

Lemma 45 *Starting from any configuration, for every alive process p, if there is a time after which $l \in LocalLeaders(p)$ forever, then there is a time after which $Leader_p = l$ forever.*

Proof. Assume that there is a time after which $l \in LocalLeaders(p)$ forever. Then, as $l \in LocalLeaders(p)$ holds infinitely often and $LocalLeaders(p) = \{LocalLeader_p[q] : q \in Alives_p\}$, there is a subset of processes V' such that:

1. $\forall q \in V'$, $q \in Alives_p$ and $LocalLeader_p[q] = l$ holds infinitely often.

Also, as there is a time t after which $l \in LocalLeaders(p)$ forever, we have the following additionnal property:

2. $\forall t' \geq t$, $\exists q_{t'} \in V'$ such that $q_{t'} \in Alives_p$ and $LocalLeader_p[q_{t'}] = l$ at time t'.

By 1. and Lemma 43, there is a time after which $\forall q \in V'$, ($LocalLeader_p[q] = l$) $\Rightarrow$ ($LocalLeaderCounter_p[q] = c_l$)) each time p sets $Leader_p$ to $MyLeader_p$. Then, by 2., there is a time t such that if p sets $Leader_p$ to $MyLeader_p$ at a time $t' \geq t$, then there exists a process $q_{t'} \in V'$ such that $LocalLeader_p[q_{t'}] = l$ and $LocalLeaderCounter_p[q_{t'}] = c_l$ at time t'.

Assume now, by the contradiction, that $Leader_p \neq l$ infinitely often. Then, as $Leader_p$ is periodically set to $MyLeader_p$ (Line 37), the following situation appears infinitely often: p sets $Leader_p$ to $MyLeader_p$ while there exists two processes v and r such that $v \in Alives_p$, $LocalLeader_p[v] = r$, and ($LocalLeaderCounter_p[v]$,$LocalLeader_p[v]$) < (c_l,l). Two case are then possible:

- $c_r < \infty$. Then, there is a time after which the condition ($LocalLeader_p[v] = r$) $\Rightarrow$ ($LocalLeaderCounter_p[v] = c_r$) holds each time p sets $Leader_p$ to $MyLeader_p$, by Part (a) of Lemma 43. Now, by Definition (c_r,r) > (c_l,l). So, ($LocalLeaderCounter_p[v]$,$LocalLeader_p[v]$) > (c_l,l) eventually holds each time p sets $Leader_p$ to $MyLeader_p$ while $v \in Alives_p$ and $LocalLeader_p[v] = r$ — a contradiction.

- $c_r = \infty$. Then, there is a time after which the condition ($LocalLeader_p[v] = r$) $\Rightarrow$ ($LocalLeaderCounter_p[v] > c_l$) holds each time p sets $Leader_p$ to $MyLeader_p$, by Part (b) of Lemma 43. So, ($LocalLeaderCounter_p[v]$,$LocalLeader_p[v]$) > (c_l,l) eventually holds each time p sets $Leader_p$ to $MyLeader_p$ while $v \in Alives_p$ and $LocalLeader_p[v] = r$ — a contradiction.

□

We now proceed to show that for every alive process p there is a time after which $l \in LocalLeaders(p)$.

Lemma 46 *Starting from any configuration, there is a time after which $l \in Alives_s$ forever.*

Proof. If $l = s$, then the lemma trivially holds. Assume now that $l \neq s$. There are three possible cases: (1) there is a time after which $l \in Alives_s$ forever, (2) l is added and removed from $Alives_s$ infinitely often, or (3) there is a time after which $l \notin Alives_s$ forever. We now show that Cases (2) and (3) cannot occur.

In case (2), l is removed from $Alives_s$ each time l was is $Alives_s$ but not in $Collect_s$ and s sets $OtherAlives_s$ to $Collect_s$ (Line 31). In this case, s sends an ACCUSATION message to l (Line 32-34). So, s sends ACCUSATION messages to l infinitely often.

In case (3), as there is a time after which $l \notin Alives_s$ forever and as s periodically sends ACCUSATION messages to every process q such that $q \in V \setminus Alives_s$, s sends ACCUSATION messages to l infinitely often.

So, in both Cases (2) and (3), s sends ACCUSATION messages to l infinitely often. Now, since the output links of s are timely and l tries to receives ACCUSATION messages from s infinitely often (exactly once by *repeat forever* loop iteration), l receives ACCUSATION messages from s infinitely often. Thus, l increments $Counter_l[l]$ infinitely often and, as $Counter_l[l]$ is monotonically nondecreasing (Observation 10), $Counter_l[l]$ unbounded — a contradiction. Hence, only Case (1) is possible. □

Lemma 47 *Starting from any configuration, there is a time after which $LocalLeader_s[s] = l$ forever.*

Proof. Immediate from Lemmas 44 and 46. □

Lemma 48 *Starting from any configuration, for every alive process p, $LocalLeader_p[s] = l$ eventually holds forever.*

Proof. Let p be an alive process. If $p = s$, then the result is immediate from Lemma 47. Assume now that $p \neq s$. In this case, p receives ALIVE messages from s infinitely often by Lemma 39. By Lemma 47, there is a time t after which $LocalLeader_s[s] = l$. So, after time t, all the ALIVE messages that s sends to p are of the form (ALIVE,l,−,−). Thus, there is a time after which all the ALIVE messages that p receives from s are of the form (ALIVE,l,−,−). So, there is a time after which $LocalLeader_p[s] = l$ forever. □

Corollary 8 *Starting from any configuration, each alive process p eventually satisfies $l \in LocalLeaders(p)$ forever.*

Proof. Immediate from Lemmas 40, 48, and Definition 9. □

Lemma 49 *Starting from any configuration, for every alive process p, there is a time after which $Leader_p = l$ forever.*

Proof. Immediate from Corollary 8 and Lemma 45. □

Theorem 10 *Algorithm 5 implements a pseudo-stabilizing leader election in System $\mathcal{S}_1$.*

Proof. Immediate from Lemma 49 and the fact that l is alive. □