Stabilizing Leader Election in Partial Synchronous Systems with Crash Failures

C. Delporte-Gallet\textsuperscript{a}, S. Devisme\textsuperscript{b,\ast}, H. Fauconnier\textsuperscript{a}

\textsuperscript{a}LIAFA UMR 7089, Université Denis Diderot, France
\textsuperscript{b}VERIMAG UMR 5104, Université Joseph Fourier, France

Abstract

This article deals with stabilization and fault-tolerance. We consider two types of stabilization: the self- and the pseudo- stabilization. Our goal is to implement the self- and/or pseudo- stabilizing leader election in systems with process crashes, weak reliability, and synchrony assumptions. We try to propose, when it is possible, communication-efficient implementations. Our approach allows to obtain algorithms that tolerate both transient and crash failures.

Note that some of our solutions are adapted from existing fault-tolerant algorithms. The motivation here is not to propose new algorithms but merely to show some assumptions required to obtain stabilizing leader elections in systems with crash failures. In particular, we focus on the borderline assumptions where we go from the possibility to have self-stabilizing solutions to the possibility to only have pseudo-stabilizing ones.

Key words: Fault-tolerance, self-stabilization, pseudo-stabilization, leader election.

1 Introduction

Self-stabilization\textsuperscript{2} is a versatile technique to design algorithms tolerating transient failures: a self-stabilizing algorithm, regardless of the initial configuration of the system, guarantees that the system reaches in a finite time a configuration $\gamma$ from which it cannot deviate from its intended behavior, i.e., every possible execution suffix starting from $\gamma$ is correct.

\*\* A preliminary version of this article was presented in SSS’07, see C. Delporte-Gallet, S. Devisme, H. Fauconnier (2007) \textsuperscript{1}.
\* Corresponding author.

Email addresses: cd@liafa.jussieu.fr (C. Delporte-Gallet), stephane.devisme@imag.fr (S. Devisme), hf@liafa.jussieu.fr (H. Fauconnier).
A weaker property called pseudo-stabilization is introduced in [3]: 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. Such an algorithm can deviate from its intended behavior an arbitrary but finite number of time, hence each of its executions has a correct suffix.

Another approach is the fault-tolerance: fault-tolerance is a property that enables a system to continue operating (possibly in a degraded mode) rather than failing completely, when some components of the system crash. One goal of this article is to obtain algorithms that are both fault-tolerant and stabilizing. For this, we study to the (eventual) leader election: this problem consists in making the system converge to a configuration from which the same alive process is forever distinguished as the leader. The leader election has been extensively studied in both stabilizing (e.g., [4,5]) and fault-tolerant (e.g., [6,7]) areas.

The stabilizing algorithms are usually not designed to withstand crash failures. Some derived approaches, generally called fault-tolerant (self- or pseudo-) stabilization have been proposed in [8–10] to design algorithms that (self- or pseudo-) stabilize even when some crashes occur dynamically in the network. But, as proved in the same articles, without very strong assertions fault-tolerant self-stabilization is impossible to ensure and concerning the leader election problem we prove that fault-tolerant self-stabilization is intrinsically impossible to solve. Hence we consider systems in which failures of processes are static. In these systems, we study self-stabilization in systems with static crash failures (SSSCF) and pseudo-stabilization in the same systems (PSSCF). We prove that considering only static process failures is not a restriction here because for the considered problem, PSSCF is equivalent to fault-tolerant pseudo-stabilization.

The impossibility results in [11,9] constraints us to make assumptions on the link timeliness. So, we look for the weakest assumptions allowing to obtain SSSCF or PSSCF leader election algorithm in fully connected networks where some processes may crash.

We also consider the communication-efficiency (denoted as CE in the rest of the article): an algorithm is communication-efficient if eventually it uses only $n - 1$ unidirectional links where $n$ is the number of processes, which is optimal [12].

We show that the notions of immediate timeliness and eventually timeliness are in some sense equivalent in the fault-tolerant stabilization. As a consequence, we only consider timeliness properties that are immediate. In the systems we study: (1) all the processes are timely and can communicate with each other
but any of them can crash and (2) some links may have timeliness or reliability properties. Our starting point is the fully timely system noted $S_4$. We show that the SSSCF leader election can be communication-efficiently implemented in $S_4$. We then show that such a strong timeliness is required in the systems we consider to obtain a CE-SSSCF leader election. Nevertheless, we also show that a SSSCF leader election that is not communication-efficient can be obtained in a weaker system ($S_3$), e.g., any system having a timely bi-source that is a process whose all input and output links are timely[13]. More generally, we show that a system having at least one path of timely links between each pair of alive processes is required to obtain the SSSCF leader election. We then consider the PSSCF. We show that a CE-PSSCF leader election can be done in some weak systems where the SSSCF leader election cannot be done: any system having a timely source that is a process whose all output links are timely[14] ($S_2$). Using a previous result of Aguilera et al [14], we then recall that communication-efficiency cannot be done if we consider systems having at least one timely source but no fair hub (a hub is a process whose all links are fair lossy) ($S_1$). However, we show that a non-communication-efficient PSSCF solution can be implemented in such systems. Finally, we conclude with the basic system where all links can be asynchronous and lossy ($S_0$): the leader election has neither SSSCF nor PSSCF solution in $S_0$ [14,9].

2 Preliminaries

2.1 Distributed Systems

Begin by recalling general definitions for stabilization in distributed systems. We model the executions of an algorithm $A$ in the system $S$ using the pair $(C,\rightarrow)$ where $C$ is the set of configurations and $\rightarrow$ is a binary transition relation on $C$. A configuration is defined as the product of the states of the processes and the state of the environment. The state of each process is the state of its local memory and the state of the environment depends on the system $S$. A step $\gamma \rightarrow \gamma'$ is either a step of some process $p$ (e.g. writing a message in a communication link) or a step of the system (e.g. delivering a message). We consider executions with time: an execution of $A$ in $S$ 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 $\forall i \geq 0$, $\gamma_i \in C$, $\gamma_i \rightarrow \gamma_{i+1}$, and the transition $\gamma_i \rightarrow \gamma_{i+1}$ occurs after $\tau_i$ time units. For each configuration $\gamma$ in $e$, $\overline{e_\gamma}$ denotes the suffix of $e$ starting in $\gamma$, conversely, $\overline{\gamma}$ denotes the associated prefix, i.e., $e = \overline{\gamma} \overline{e_\gamma}$. More generally, given configuration $\gamma$, $\overline{\gamma}$ denotes the set of all suffixes for all the executions beginning in configuration $\gamma$.

A specification is a predicate over the executions. Let $A$ be an algorithm in the system $S$. Let $F$ be a specification, and given some set $I$ of initial configurations we say that algorithm $A$ satisfies $F$ for $I$ in $S$ if all the executions beginning in configurations belonging to $I$ satisfy $F$.

For stabilizing properties, we consider that the executions can start from any
configuration.

\( \mathcal{A} \) is \textit{self-stabilizing} for \( \mathcal{F} \) in \( \mathcal{S} \) if and only if in any execution of \( \mathcal{A} \) in \( \mathcal{S} \), there exists a configuration \( \gamma \) such that all possible suffixes in \( \gamma \) satisfy \( \mathcal{F} \).

\( \mathcal{A} \) is \textit{pseudo-stabilizing} for \( \mathcal{F} \) in \( \mathcal{S} \) if and only if in any execution of \( \mathcal{A} \) in \( \mathcal{S} \), there exists a suffix that satisfies \( \mathcal{F} \).

2.2 Fault-Tolerance and Stabilization

Given system \( \mathcal{S} \), to deal with crashes, we define \( \mathcal{S}_{\text{crash}} \) in such a way that the environment of any configuration gives the set of processes that are alive and a step of a process is possible only if this process is alive. By definition a process that is not alive is said \textit{crashed} or \textit{dead}. More precisely, if \( \gamma \) is a configuration then \( A(\gamma) \) denotes the set of processes that are alive in state \( \gamma \). Only a process that is alive can make a step: if \( \gamma \rightarrow \gamma' \) by a step of process \( p \) in \( \mathcal{S} \), then \( p \in A(\gamma) \). Moreover there is no reparation and a process dead is dead forever: if \( p \notin A(\gamma) \), then for all \( \gamma' \) such that \( \gamma \rightarrow \gamma' \), \( p \notin A(\gamma') \).

Classically, stabilizing algorithms withstand the transient faults because, after such failures, the system can be in an arbitrary configuration and, in this case, a stabilizing algorithm guarantees that the system recovers a correct behavior in a finite time without any external intervention if no transient fault appears during this convergence. To show the stabilization, we observe the system from the first configuration after the end of the last transient fault, yet considered as the \textit{initial configuration} of system. Actually, if we prove that from such a configuration, an algorithm guarantees that the system recovers a correct behavior in a finite time, it means that the algorithm guarantees that the system will recover if the time between two periods of transient faults is sufficiently large. Henceforth, such an algorithm can be considered as tolerating transient faults.

Here, as in [8–10], we not only consider transient faults: our systems may go through transient as well as crash failures. A crashed process definitively stops to execute its local algorithm. In this way crashes are not transient failures. In order to deal with this kind of failures we consider two types of systems, where in the first ones process failures are static and in the second ones processes may crash dynamically.

In the first ones, we denote \( \mathcal{S}^{\text{static}} \), we consider the crashes in the same way that transient faults. That is, we consider the first configuration after the last crash as the \textit{initial configuration} of the system. Hence, given \( \mathcal{S}_{\text{crash}} \), for \( \mathcal{S}^{\text{static}} \) we have: if \( \gamma \rightarrow \gamma' \) then \( A(\gamma) = A(\gamma') \).

We say that algorithm \( \mathcal{A} \) is (self- pseudo-) \textit{stabilizing for} \( \mathcal{F} \) for system \( \mathcal{S} \) with \textit{static crash failures} if and only if \( \mathcal{A} \) is (self- pseudo-) stabilizing for \( \mathcal{F} \) in
As crashes do not occur dynamically, any process that is alive in the initial configuration is alive forever. Any subset of processes may be crashed in the initial configuration. The fact that we consider only initial crashes corresponds to the classical stabilizing approach. Our fault-tolerant stabilizing algorithms guarantee that if the time between two periods of failures — these periods can contain an arbitrary number of process crashes and transient failures — is sufficiently large, then the system eventually recovers.

If we do not restrict to initial crashes, we have systems $S_{dyn}$ in which in any configuration any alive process may crash. More precisely, given $S_{crash}$, we define $S_{dyn}$ in such a way that for each $\gamma$, for each $p \in A(\gamma)$ there is a step $\gamma \mapsto \gamma'$ such that $\gamma$ and $\gamma'$ differs only concerning the set of processes that are alive and $A(\gamma') = A(\gamma) - \{p\}$.\footnote{We consider here that there is no restriction on the process that may crash, it is straightforward to extend these definitions for systems with some restrictions on the set of crashed process (e.g. systems with a majority of correct processes).}

If we do not consider transient faults, given some set $I$ of initial configurations, we get the classical definition of fault-tolerance: an algorithm $A$ is fault-tolerant for $F$ with initial configurations $I$ if and only if $A$ satisfies $F$ in system $S_{dyn}$ for $I$.

As usual, stabilization consider that the executions can start from any configuration and then all configurations are considered as being initial. We say that algorithm $A$ is fault-tolerant (self- pseudo-) stabilizing for $F$ in system $S$ if $A$ is (self- pseudo-) stabilization in $S_{dyn}$.

Clearly, fault-tolerant stabilization is stronger than stabilization for systems with static crash failures

**Observation 1** If $A$ is fault-tolerant (self- pseudo-) stabilizing for $F$ in system $S$, then it is also (self- pseudo-) stabilizing for $F$ in system $S$ with static crash failures.

### 2.3 Leader Election

In this article we are interested in the leader election problem. In this problem, each process $p$ has a variable $Leader_p$ that holds the identity of a process. The leader election has to ensure that (1) all alive processes should hold the identity of the same process forever and (2) that this process is alive forever. More precisely, $q$ is a leader in configuration $\gamma$ if every alive process $p$ has its $Leader_p$ variable set to $q$. Given an execution $e$ and $\gamma$ a configuration in $e$, $q$
is elected in $\gamma$ for $e$ if and only if $q$ is leader in $\gamma$ and for every configuration $\gamma'$ in $\overline{\gamma}^\gamma$, $q$ is elected in $\gamma'$ and $q \in A(\gamma')$.

Then the predicate on execution $e$ of the leader election for stabilization is: some process is elected in the first configuration of $e$.

Let $A$ be a pseudo-stabilizing algorithm for leader election, let $e$ any execution of $A$ in $S^{\text{dyn}}$, there is some suffix $f$ of $e$ such that there is no new crash in $f$, let $\gamma$ be the first configuration in $f$, then $f$ is an execution of $A$ in $S^{\text{static}}$ and there is some suffix $g$ in $f$ that ensure the election of some process, and then for $e$ there is some suffix for which some process is elected, proving that $A$ is also fault-tolerant pseudo-stabilizing. Then we have:

**Observation 2** $A$ is fault-tolerant pseudo-stabilizing for leader election if and only if $A$ is pseudo-stabilizing for leader election in system with static crash failures.

Classically, in fault-tolerance, we consider the eventual leader election problem defined by the predicate: in any execution $e$ from initial state there is some configuration $\gamma$ in $e$ such that some process is elected in $\gamma$ for $e$. From a similar argument to Observation 2, a pseudo-stabilizing algorithm for the leader election problem in system with static crash failures is also fault-tolerant algorithm for the eventual leader election for every initial configuration.

**Observation 3** If $A$ is self- or pseudo- stabilizing for leader election in systems with static crash failures or $A$ is fault-tolerant self- or pseudo- stabilizing for leader election then $A$ is fault-tolerant for the eventual leader election for every initial configurations.

But, concerning fault-tolerant self-stabilizing, remark that there is no fault-tolerant self-stabilizing algorithm for the leader election: by contradiction, assume that there is such an algorithm, and consider some execution with at least two processes, say $p$ and $q$, that are alive forever, assume that there is some configuration for which $p$ is elected, then in $S^{\text{dyn}}$, $p$ may crash in $\gamma$ and then we get a suffix in $\overline{\gamma}$ for which $p$ is not elected. Then there is no configuration in which some process is elected in all suffixes. Hence we have:

**Observation 4** There is no fault-tolerant self-stabilizing algorithm for leader election.

Observations 4 and 2 prove that the properties of fault-tolerant stabilization may be deduced from pseudo-stabilization for systems with static crash failures. Then for the leader election problem only self-stabilization and pseudo-stabilization for system with static crash failures (denoted by SSSCF and PSSCF) are interesting and in the following we consider only them.
2.4 Message-Passing Model

2.4.1 Processes

The processes execute their program by taking atomic 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.

We assume that there exists a known non-null lower bound on the time required by the alive processes to execute a step. 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. Our algorithms are structured as a repeat forever loop in which each process executes only a bounded number of steps in each loop iteration. Hence, each alive process satisfies a lower and an upper bound, noted $\alpha$ and $\beta$, respectively, on the time it requires to execute a loop iteration. We assume that $\alpha$ and $\beta$ are known by each process. Without loss of generality, we also assume that $\alpha = 1$ and $\beta \in \mathbb{N}^*$.

Observation 5 For every time $t$, an alive process $p$ executes at least one complete loop iteration during the time interval $[t, t + 2\beta]$.

2.4.2 Links

The processes can send messages over directed links. There is a directed link from each process to all the others. A message $m$ carries a type $T$ and a data $D$. For each incoming link $(q,p)$ and each type $T$, the process $p$ has a buffer, $\text{Buffer}_p[q,T]$, that can hold one single message of type $T$. $\text{Buffer}_p[q,T] = \perp$ when it holds no message. If $q$ sends a message $m$ to $p$ and the link $(q,p)$ does not lose $m$, $\text{Buffer}_p[q,T]$ is eventually set to $m$. If it happens, we say that message $m$ is delivered to $p$ from $q$. If $\text{Buffer}_p[q,T]$ was set to some previous message, this message is overwritten. When $p$ takes a step, it may choose a process $q$ and a type $T$ to read the content of $\text{Buffer}_p[q,T]$. If $\text{Buffer}_p[q,T]$ contains a message $m$, we say that $p$ receives $m$ from $q$ and $\text{Buffer}_p[q,T]$ 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.

A fair link $(p,q)$ satisfies: for each type of message $T$, if $p$ sends infinitely many messages of type $T$ to $q$, 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 $p$. A link $(p,q)$ is timely if there exists a constant $\delta$ 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 $\delta$). A link $(p,q)$ is eventual timely if there exists a constant $\delta$ 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 eventual timely link
Systems | Properties
--- | ---
$S_0$ | Links: arbitrary slow, lossy, and initially not necessarily empty
Processes: can be initially crashed, timely forever otherwise
Variables: initially arbitrary assigned
$S_1$ | $S_0$ with at least one timely source
$S_2$ | $S_0$ with at least one timely source and at least one fair hub
$S_3$ | $S_0$ with at least one timely bi-source
$S_4$ | $S_0$ with all links timely

Fig. 1. Systems considered in this article ($\forall i, 0 < i \leq 4, S_i \subset S_{i-1}$).

at time $t$ is delivered within time $t + \delta$). We assume that every process knows $\delta$ and without loss of generality, $\delta$ is a multiple of $\beta$: let $\sigma \in \mathbb{N}^*, \delta = \sigma \times \beta$.

2.4.3 Particular Characteristics

A timely source (resp. an eventual timely source) [14] is a timely process having all its output links that are timely (resp. eventual timely). A timely bi-source (resp. an eventual timely bi-source) [13] is a timely process having all its (input and output) links that are timely (resp. eventual timely). The digraph $G = (V, E)$ is a timely routing overlay (resp. eventual timely routing overlay) if $V$ is the set of all timely processes, $E$ only contains timely (resp. eventual timely) links, and $G$ is strongly connected. A fair hub [14] is an alive process having all its (input and output) links that are fair.

2.4.4 Systems

All our 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 arbitrary messages, and (3) except if we explicitly state, there is no assumption on the fairness and the timeliness of the links.

The system $S_0$ corresponds to the system where no further assumptions are made: in $S_0$, the links can be arbitrary slow or lossy. In $S_1$, we assume that there exists at least one timely source whose identifier is unknown. In $S_2$, we assume that there exists at least one timely source and at least one fair hub. The timely source and the fair hub can be the same process or not, their identifiers are unknown. In $S_3$, we assume that there exists at least one timely bi-source whose identifier is unknown. In $S_4$, all links are timely. Figure 1 summarizes the properties of our systems.

Remark 1 Any variable can be arbitrary assigned in the initial configuration. In particular, the program counter may not point to the first instruction in the initial configuration. This may cause some problems in the first loop iteration: if the program counter initially points to an instruction in an if block, the instruction is executed without the guarantee that the test of the if is true.
However, such a problem can only appear during the first loop iteration.

2.4.5 Timeliness vs. Eventual Timeliness

Theorems 1 and 2 justify why we use the timeliness instead of eventual timeliness.

Below we use the following notations: Let $A$ be an algorithm. Let $F$ be a specification. Let $S$ be a system having some timely links. Let $S'$ be a system having some eventual timely links such that a link is timely in $S$ if and only if this link is eventual timely in $S'$.

**Theorem 1** $A$ is PSSCF for $F$ in $S$ if and only if $A$ is PSSCF for $F$ in $S'$.

**Proof.** Since the if part is trivial by definition, we focus on the only if part: Assume, by contradiction, that $A$ is PSSCF for $F$ in $S$ but not PSSCF for $F$ in $S'$. Then, there exists an execution $e$ of $A$ in $S'$ such that no suffix of $e$ satisfies $F$. Let $\gamma$ be the configuration of $e$ from which all the eventual timely links of $S'$ are timely. As no suffix of $e$ satisfies $F$, no suffix of $e\gamma$ satisfies $F$ too. Now, $e\gamma$ is a possible execution of $A$ in $S$ because (1) $\gamma$ is a possible initial configuration of $S$ and (2) every eventual timely link of $S'$ is timely in $e\gamma$. Hence, as no suffix of $e\gamma$ satisfies $F$, $A$ is not PSSCF for $F$ in $S$ — a contradiction. \qed

Following similar arguments, we have:

**Theorem 2** $A$ is SSSCF for $F$ in $S$ if and only if $A$ is SSSCF for $F$ in $S'$.

3 System $S_4$

We first consider System $S_4$. From [8], we already know that SSSCF leader election algorithms can be implemented in such systems. Here we show that SSSCF can be not only but communication-efficiently achieved in those systems.

3.1 CE-SSSCF Leader Election in $S_4$

The goal of our algorithm, Algorithm 1, is to elect the alive process with the smallest identifier.

To obtain the communication-efficiency, Algorithm 1 uses only one message type (ALIVE) and proceeds as follows:

(1) A process $p$ periodically sends ALIVE messages containing its own identifier only if it believes to be the leader, i.e., only if $\text{Leader}_p = p$.

Using this mechanism, Algorithm 1 is communication-efficient because, since the system is stabilized, there is only one leader. Note that the leader must
periodically send ALIVE messages to be not suspected of being crashed.

We want that the smallest alive process $\ell$ eventually satisfies $\text{Leader}_\ell = \ell$ forever. To that goal, we use the following principle:

(2) If a process $p$ receives no (ALIVE) message from $q$ such that $q < p$ and $q \leq \text{Leader}_p$ during a well-chosen period of time, then it starts to believe to be the leader, i.e., $\text{Leader}_p \leftarrow p$.

Thanks to (2), the smallest alive process $\ell$ eventually satisfies $\text{Leader}_\ell = \ell$ even if $\text{Leader}_\ell$ is initially assigned to a smaller identifier of a crashed or non-existing process. Moreover, at that time $\ell$ starts sending ALIVE message according to (1) and, thanks to (2), every other alive process $p$ eventually satisfies $\text{Leader}_p \geq \ell$ forever. It remains now to guarantee that all other alive processes eventually definitely adopt $\ell$ as leader. To that goal, we proceed as follows:

(3) When a process $p$ receives an (ALIVE) message from $q$, $p$ sets $\text{Leader}_p$ to $q$ if $q < p$ and $q \leq \text{Leader}_p$.

To implement principles (1) and (2), we uses two counters: $\text{SendTimer}_p$ and $\text{ReceiveTimer}_p$. These counters are incremented at each loop iteration in order to evaluate a particular time interval. Using the lower and upper bound on the time to execute an iteration of this loop, each process $p$ knows how many iterations it must execute before a given time interval passed.

```
Algorithm 1 CE-SSSCF Leader Election in $S_4$, code for every process $p$

1: variables:
2: \hspace{0.5cm} $\text{Leader}_p \in \{1, \ldots, n\}$
3: \hspace{0.5cm} $\text{SendTimer}_p, \text{ReceiveTimer}_p$: non-negative integers
4: repeat forever
5: \hspace{0.5cm} for all $q \in V \setminus \{p\}$ do
6: \hspace{1.0cm} if receive(ALIVE) from $q$ then
7: \hspace{1.5cm} if $(q < p) \land (q \leq \text{Leader}_p)$ then
8: \hspace{2.0cm} $\text{Leader}_p \leftarrow q$
9: \hspace{2.0cm} $\text{ReceiveTimer}_p \leftarrow 0$
10: \hspace{1.5cm} end if
11: \hspace{1.0cm} end if
12: \hspace{0.5cm} end for
13: $\text{SendTimer}_p \leftarrow \text{SendTimer}_p + 1$
14: \hspace{0.5cm} if $\text{SendTimer}_p \geq \sigma$ then
15: \hspace{1.0cm} if $\text{Leader}_p = p$ then
16: \hspace{1.5cm} send(ALIVE) to every process except $p$
17: \hspace{1.5cm} end if
18: \hspace{1.0cm} $\text{SendTimer}_p \leftarrow 0$
19: \hspace{0.5cm} end if
20: $\text{ReceiveTimer}_p \leftarrow \text{ReceiveTimer}_p + 1$
21: \hspace{0.5cm} if $\text{ReceiveTimer}_p > 5\delta$ then
22: \hspace{1.0cm} $\text{Leader}_p \leftarrow p$
23: \hspace{1.0cm} $\text{ReceiveTimer}_p \leftarrow 0$
24: \hspace{1.0cm} end if
25: end repeat
```
3.2 Correctness of Algorithm 1 in $S_4$

We recall that in the following proofs, we assume that the initial configuration of the system is arbitrary. Also, we will denote by $\ell$ the alive process with the smallest identifier.

**Lemma 1** For every alive process $p$, if $p$ receives a message $m$ from $q$ at time $t > \delta + 2\beta$, then $q$ is alive and $q$ sends $m$.

**Proof.** First, as we only consider initial crashes, $p$ can only receive messages that are either initially in a link or sent by alive processes. Hence, to prove the lemma, we just have to show that $p$ cannot receive a message that was initially in a link after time $\delta + 2\beta$. Now, after $\delta$ times, all messages initially in the links are delivered. Moreover, at every loop iteration $p$ tries to receive a message for each other process and $p$ executes a complete loop iteration at least every $2\beta$ times. Hence, after time $\delta + 2\beta$, $p$ can only receive messages that were sent by alive processes and the lemma holds. $\square$

According to Remark 1 (page 8), we have the following observation:

**Observation 6** A process $p$ sets $\text{Leader}_p$ to $q \neq p$ at time $t > \beta$ (Line 8) only if $(q < p)$ and $(q \leq \text{Leader}_p)$ at time $t$.

**Lemma 2** $\text{Leader}_\ell = \ell$ at every time $t$ such that $t > (5\beta + 1)\delta + 4\beta$.

**Proof.** Let $t > \delta + 2\beta$. From time $t$, any ALIVE message that $\ell$ receives was sent by an alive process $q$ such that $q > \ell$ by Lemma 1 and the definition of $\ell$. Hence, if $\text{Leader}_\ell = \ell$ holds, then $\text{Leader}_\ell = \ell$ holds forever by Observation 6. Assume now that $\text{Leader}_\ell \neq \ell$. Then, $\ell$ points the first instruction of the loop, at most at time $t + \beta$. From that time, while $\text{ReceiveTimer}_\ell \leq 5\delta$, $\ell$ increments $\text{ReceiveTimer}_\ell$ at each loop iteration and each loop iteration is performed in at most $\beta$ times. Hence, at the latest in the $5\delta + 1^{th}$ loop iteration from time $t + \beta$, $\ell$ satisfies the test of Line 21, sets $\text{Leader}_\ell$ to $\ell$ (Line 22), and then $\text{Leader}_\ell = \ell$ holds forever by Lemma 1, the definition of $\ell$, and Observation 6. Hence, $\text{Leader}_\ell = \ell$ holds forever after at most $(5\delta + 1) \times \beta$ from time $t + \beta$ with $t > \delta + 2\beta$, and the lemma holds. $\square$

Similar to Lemma 2, we obtain the following lemma:

**Lemma 3** For every alive process $p$, $\text{Leader}_p \geq \ell$ at every time $t$ such that $t > (5\beta + 1)\delta + 4\beta$.

**Lemma 4** For every time $t > (5\beta + 1)\delta + 4\beta$, $\ell$ sends an ALIVE message to every other process during the time interval $[t, t + \delta + \beta]$. 11
Proof. Let \( t \) be any time such that \( t > (5\beta + 1)\delta + 4\beta \). From time \( t \), the program counter of \( \ell \) points to the first loop instruction at time \( t' \leq t + \beta \). From time \( t' \), \( \ell \) executes a complete loop iteration at most every \( \beta \) times. Also, from time \( t' \), while \( SendTimer_\ell < \sigma \), \( SendTimer_\ell \) is incremented at each loop iteration. So, as \( SendTimer_\ell \) is always greater or equal to 0, \( SendTimer_\ell \geq \sigma \) becomes true at the latest during the \( \sigma \)th loop iteration from time \( t' \) and \( \ell \) sends ALIVE to any other process in the same loop iteration because \( Leader_\ell = \ell \) (Lemma 2). Hence, from time \( t' \), \( \ell \) sends ALIVE to every other process in at most \( \sigma \times \beta \) times, i.e., in at most \( \delta \) times. As \( t' \leq t + \beta \), the lemma is proven. \( \square \)

Lemma 5  For every alive process \( p \), \( Leader_p = \ell \) at every time \( t \) such that \( t > (5\beta + 3)\delta + 8\beta \).

Proof. Let \( t > (5\beta + 1)\delta + 4\beta \). We prove the lemma in two steps:
(1) We first prove that if \( p \) receives an ALIVE message from \( \ell \) at time \( t' > t \), then \( Leader_p \geq \ell \) holds forever from time \( t' + \beta \).
(2) We then prove that \( p \) receives an ALIVE message from \( \ell \) at most at time \( t + 2\delta + 3\beta \).

Proof of (1): Assume that \( p \) receives an ALIVE message from \( \ell \) at time \( t' > t \). Then, from the code of Algorithm 1 and Lemma 3, \( p \) satisfies \( Leader_p = \ell \) and \( ReceiveTimer_p = 0 \) at the end of the loop iteration, i.e., before time \( t' + \beta \). Assume, by the contradiction, that \( Leader_p \neq \ell \) eventually holds after time \( t' + \beta \). Then, this means that \( p \) executes at least \( 5\delta \) consecutive complete loop iterations, i.e. at least \( 5\delta \) times, without any ALIVE message from \( \ell \) was delivered to \( p \). By Lemma 4 and the fact that any message is delivered at most \( \delta \) times after its sending, we obtain a contradiction. Hence (1) is proven.

Proof of (2): By Lemma 4, \( \ell \) sends an ALIVE message to \( p \) during the time interval \( [t, t + \delta + \beta] \). Then, this message is delivered to \( p \) at most \( \delta \) times after. As \( p \) tries to receive a message from \( \ell \) at each loop iteration (at most every \( 2\beta \) times), \( p \) receives an alive message from \( \ell \) before time \( t + 2\delta + 3\beta \), which completes the proof. \( \square \)

Theorem 3  Algorithm 1 implements a CE-SSSCF leader election in system \( S_4 \).

Proof. By Lemma 5, starting from any configuration, the system reaches in a bounded time a configuration \( \gamma \) from which there is an alive process that is the unique leader forever: after that time, the system cannot deviate from its specification whatever the execution suffix, i.e., Algorithm 1 is a SSSCF leader election algorithm. Also, once the system is stabilized, only one process, \( \ell \), sends messages: Algorithm 1 is communication-efficient. \( \square \)
4 System $S_3$

In the previous section, we saw that CE-SSSCF leader election can be implemented in $S_4$. We now show that such a strong system is required to implement a leader election that is both SSSCF and communication-efficient: CE-SSSCF leader election cannot be solved in $S_3$. However, we will show that a non-communication-efficient SSSCF leader election can be implemented in $S_3$.

4.1 Impossibility of the CE-SSSCF Leader Election in $S_3$

To prove this impossibility, we show that no CE-SSSCF leader election can be implemented in $S_4^-$: any system $S_0$ having (1) all its links that are reliable and (2) all its links that are timely except at most one which can be asynchronous.

Lemma 6 Let $A$ be a SSSCF leader election algorithm in $S_4^-$. In every execution of $A$, every alive process $p$ satisfies: from any configuration where $\text{Leader}_p \neq p$, $\exists k \in \mathbb{R}^+$ such that $p$ modifies $\text{Leader}_p$ if it receives no message during $k$ times.

Proof. Assume, by contradiction, that there exists an execution $e$ where there is a configuration $\gamma$ from which a process $p$ satisfies $\text{Leader}_p = q$ forever with $q \neq p$ while $p$ receives message forever. As $A$ is SSSCF, it can start from any configuration. So, $e_\gamma$ is a possible execution. Let $\gamma'$ be a configuration which is identical to $\gamma$ except that $q$ is crashed in $\gamma'$. Consider any execution $e_{\gamma'}$ starting from $\gamma'$ where $p$ receives no message forever. As $p$ cannot distinguish $e_\gamma$ and $e_{\gamma'}$, it behaves in $e_{\gamma'}$ as in $e_\gamma$: it keeps $q$ as leader while $q$ is crashed — a contradiction. □

Theorem 4 There is no CE-SSSCF leader election algorithm in $S_4^-$. 

Proof. Assume, by contradiction, that there exists a CE-SSSCF leader election algorithm $A$ in $S_4^-$. Consider an execution $e$ where no process crashes and all the links behave as timely. By definition of self-stabilization and Lemma 6, there exists a configuration $\gamma$ in $e$ such that in any suffix starting from $\gamma$: (1) there exists an alive process $\ell$ such that any alive process $p$ satisfies $\text{Leader}_p = \ell$ forever, and (2) messages are received infinitely often through at least one input link of each alive process except perhaps $\ell$.

Let $e_\gamma$ be the suffix of $e$ where every alive process $p$ satisfies $\text{Leader}_p = \ell$ forever. Communication-efficiency and (2) implies that messages are received infinitely often in $e_\gamma$ through exactly $n - 1$ links of the form $(q,p)$ with $p \neq \ell$. 

Let $E$ be the set of the $n - 1$ links where messages are sent infinitely often in $\bar{e}_{\gamma}^\ast$.

Consider now an execution $e'$ identical to $e$ except that there is a time after which some 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 $\bar{e}^\ast e''$ with $e''$ a non-empty prefix of $\bar{e}_{\gamma}^\ast$. Now, after any prefix $\bar{e}^\ast e''$, $(q, p)$ can start to arbitrary delay the messages and, in this case, $p$ eventually changes its leader by Lemma 6. Hence, for any prefix $\bar{e}^\ast e''$, there is a possible suffix of execution in $S_4^-$ where $p$ changes its leader: this contradicts the definition of self-stabilization.

By definition, any system $S_4^-$ having $n \geq 3$ processes is a system $S_3$. Hence:

**Corollary 1** There is no CE-SSSCF leader election algorithm in $S_3$ for $n \geq 3$ processes.

### 4.2 SSSCF Leader Election in $S_3$

**Algorithm 2** SSSCF Leader Election on $S_3$, code for every process $p$

1: variables:
2: $\text{Leader}_p \in \{1, \ldots, n\}$
3: $\text{SendTimer}_p$, $\text{ReceiveTimer}_p$: non-negative integers
4: $\text{Collect}_p$, $\text{OtherAlives}_p$: sets of non-negative integers
5: macro:
6: $\text{Alives}_p = \text{OtherAlives}_p \cup \{p\}$ /* this macro is just used to simplify the code */
7: repeat forever
8: for all $q \in V \setminus \{p\}$ do
9: if receive($\text{ALIVE}_r$) from $q$ then
10: $\text{Collect}_p \leftarrow \text{Collect}_p \cup \{r\}$ /* $p$ collects the IDs of the alive processes */
11: if $q = r$ then
12: send($\text{ALIVE}_r$) to every process except $p$ and $q$ /* relay */
13: end if
14: end if
15: end for
16: $\text{SendTimer}_p \leftarrow \text{SendTimer}_p + 1$
17: if $\text{SendTimer}_p \geq \sigma$ then
18: send($\text{ALIVE}_p$) to every process except $p$ /* $p$ periodically sends $\text{ALIVE}_p$ */
19: $\text{SendTimer}_p \leftarrow 0$
20: end if
21: $\text{ReceiveTimer}_p \leftarrow \text{ReceiveTimer}_p + 1$
22: if $\text{ReceiveTimer}_p > 3\delta + 6\beta$ then /* $p$ periodically computes its leader */
23: $\text{OtherAlives}_p \leftarrow \text{Collect}_p$
24: $\text{Leader}_p \leftarrow \min(\text{Alives}_p)$
25: $\text{Collect}_p \leftarrow \emptyset$
26: $\text{ReceiveTimer}_p \leftarrow 0$
27: end if
28: end repeat

Algorithm 2 implements a SSSCF leader election in $S_3$. In the algorithm, each process locally computes in an $\text{Alives}$ set the list of all alive processes and designates as leader the smallest process of the set. The $\text{Alives}$ sets are computed in two steps. First, each process $p$ regularly sends $\text{ALIVE}_p$ messages through all its links. Secondly, each message is relayed once: when receiving
an ALIVE-r message from a process q, the process p retransmits it to all the other processes (except q) only if q = r. Using this method, any alive process p regularly receives an ALIVE-q message for each alive process q ≠ p within a bounded period of time. Hence, each process p can periodically evaluate in a Collect_p set the IDs of every other alive process: the IDs contained in all the messages it recently received. Eventually, the IDs of every crashed process disappear forever from the Collect sets because each message is relayed at most once. Moreover, the timely bi-source guarantees that the IDs of each other alive process are put into the Collect sets of all alive processes every bounded period of time. Hence, by periodically assigning the content of Collect_p to the set OtherAlives_p, OtherAlives_p converges to the set of all the alive processes different of p. Finally, p just has to periodically output the smallest ID in the set Alives_p = OtherAlives_p ∪ {p} so that the system converges to an unique leader.

4.3 Correctness of Algorithm 2 in S_3

We recall that in the following proofs, we assume that the initial configuration of the system is arbitrary.

**Lemma 7** Every alive process eventually no more receives ALIVE-q messages where q is a crashed process.

**Proof.** The lemma holds because every ALIVE-q message is relayed at most once and the (initially) crashed processes never send messages. □

**Lemma 8** An alive process p sends ALIVE-p to all other processes at least every δ + β times.

**Proof.** Any alive process p sends ALIVE-p to all other processes every σ complete loop iteration. p starts its first complete loop iteration after at most β times and then executes a complete loop iteration at most every β times. Hence, p sends ALIVE-p to all other processes at most every σ × β + β times, i.e., at most every δ + β times. □

**Lemma 9** Let p and q be two alive processes such that p ≠ q, p receives an ALIVE-q message at least every 3δ + 6β times.

**Proof.** The two following claims prove the lemma:

(1) p receives an ALIVE-q message from q at least every 2δ + 3β times if q or p is the timely bisource.

**Claim Proof:** q sends an ALIVE-q message to p at least every δ + β times by Lemma 8. As p or q is the timely bi-source, the link (q,p) is a timely one, i.e., any message in this link is delivered in at most δ times. Finally, every message is received at most one loop iteration after its delivery, i.e., 2β times by Observation 5 (page 7).
p receives an ALIVE-\(q\) message at least every \(3\delta + 6\beta\) times if neither \(q\) nor \(p\) are the timely bisource.

**Claim Proof:** Let \(b\) be the timely bi-source, \(b\) receives an ALIVE-\(q\) message from \(q\) at least every \(2\delta + 3\beta\) times (1). After each reception of ALIVE-\(q\) messages from \(q\), \(b\) sends ALIVE-\(q\) to \(p\) in the same loop iteration, \(i.e.,\) within \(\beta\) times. As the link \((b,p)\) is timely, any message in this link is delivered in at most \(\delta\) times. Finally, every message is received at most one loop iteration after its delivery, \(i.e.,\) \(2\beta\) times by Observation 5 (page 7).

\(\square\)

**Lemma 10** For every alive process \(p\), \(Alives_p\) is eventually equal to the set of all alive processes forever.

**Proof.** The two following claims prove the lemma:

1. Eventually \(Alives_p\) only contains IDs of alive processes.

   **Claim Proof:** Immediate from Lemma 7.

2. \(Alives_p\) eventually contains the IDs of every alive process \(q\) forever.

   **Claim Proof:** If \(p = q\), the claim trivially holds. So, consider that \(p \neq q\). In the algorithm, \(p\) periodically resets \(Collect_p\) to \(\emptyset\). After \(p\) resets \(Collect_p\), \(p\) resets \(ReceiveTimer_p\) to 0, and waits at least \(3\delta + 6\beta + 1\) loop iterations, \(i.e.,\) at least \(3\delta + 6\beta + 1\) times, before setting \(OtherAlives_p\) to \(Collect_p\). During this period, \(p\) receives at least one ALIVE-\(q\) message for \(q\) by Lemma 9 and inserts \(q\) into \(Collect_p\), which proves the claim.

\(\square\)

From Lemma 10, we can deduce:

**Theorem 5** Algorithm 2 implements a SSSCF leader election in \(S_3\).

5 **System \(S_2\)**

We saw that SSSCF leader election can be done in \(S_3\). We now show that this result is due to the existence of an eventual timely routing overlay. Indeed, we now prove that SSSCF leader election cannot be achieved in a system that does not contain an eventual timely routing overlay. Hence, it is also impossible to implement a SSSCF leader election in \(S_2\). However, we will show that a CE-PSSCF leader election can be done in \(S_2\). The algorithm we propose — Algorithm 3 — is an adaptation of an algorithm provided in [14]. It is important to note that any system \(S_3\) is also a system \(S_2\): a timely bi-source is both a source and a fair hub. Hence, Algorithm 3 also implements a CE-PSSCF leader election in \(S_3\).
5.1 Impossibility of the SSSCF Leader Election in $S_2$

To prove this impossibility, we show that no CE-SSSCF leader election can be implemented in a particular case of $S_2$: let $S_{3}^-$ be any system $S_2$ having all its links that are reliable but containing no eventually timely overlay.

Let $m$ be a message sent at 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 time $t'$ such that $t' < t$. We call causal sequence any sequence $p_0,m_1,\ldots,p_{k-1},m_k$ 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 $m_i$. In this case, we say that $m_k$ causally depends on $p_0$. We also say that $m_k$ is a new message that causally depends on $p_0$ if and only if there exists two causal sequences $p_0,m_1,\ldots,p_{k-1},m_k$ and $p_0,m_{1'},\ldots,p_{k'-1},m_{k'}$ such that $m_{1'}$ is older than $m_1$.

**Lemma 11** Let $A$ be a SSSCF leader election algorithm in $S_{3}^-$. In every execution of $A$, every alive process $p$ satisfies: from any configuration where $Leader_p \neq p$, $\exists k \in \mathbb{R}^+$ such that $p$ changes its leader if it receives no new message that causally depends on $Leader_p$ during $k$ times.

**Proof.** Assume, by contradiction, that there exists an execution $e$ where there is a configuration $\gamma$ from which a process $p$ receives no new message that causally depends on $q \neq p$ while satisfying $Leader_p = q$ forever. As $A$ is SSSCF, it can start from any configuration. So, $\gamma^e_p$ is a possible execution of $A$. Let $\gamma'$ be a configuration that is identical to $\gamma$ except that $q$ is crashed in $\gamma'$. As $p$ only received the messages that do not causally depend on $q$ in $\gamma^e_p$, there exists a possible execution $\gamma^e_p$ starting from $\gamma'$ where $p$ received exactly the same messages as in $\gamma^e_p$. Hence, $p$ cannot distinguish $\gamma^e_p$ and $\gamma^e_p$ and $p$ behaves in $\gamma^e_p$ as in $\gamma^e_p$: it keeps $q$ as leader forever while $q$ is crashed: $A$ is not a SSSCF leader election algorithm — a contradiction. $\square$

**Theorem 6** There is no SSSCF election algorithm in system $S_{3}^-$. 

**Proof.** Assume, by contradiction, that there exists a SSSCF leader election algorithm $A$ in system $S_{3}^-$. By definition of self-stabilization, in any execution of $A$, there exists a configuration $\gamma$ such that in any suffix starting from $\gamma$ there exists an unique leader and this leader no more changes. Let $e$ be an execution of $A$ where no process crashes and every link is timely. Let $\ell$ be the alive process which is eventually elected in $e$. Consider now any execution $e'$ identical to $e$ except that there is a time after which there is at least one link in each path from $\ell$ to some process $p$ that arbitrary delays the messages. Then, $e$ and $e'$ can have an arbitrary large common prefix. Hence, it is possible to construct executions of $A$ beginning with any prefix of $e$ where $\ell$ is eventually elected but in the associated suffix, any causal sequence of messages from $\ell$ to $p$ is arbitrary delayed and, by Lemma 11, $p$ eventually changes its leader.
to a process $q \neq \ell$. Thus, for any prefix $\overline{c}$ of $e$ where a process is eventually
elected, there exists a possible execution having $\overline{c}$ as prefix and an associated
suffix $\overline{e}$ in which the leader eventually changes: this contradicts the definition
of self-stabilization.

By Theorem 6, follows:

**Corollary 2** There is no SSSCF leader election algorithm in system $S_2$.

### 5.2 CE-PSSCF Leader Election in $S_2$

Our algorithm (Algorithm 3) uses the same principle as Algorithm 1 to obtain
the communication-efficiency : each process periodically sends ALIVE to all
other processes only if it thinks to be the leader. Using this principle, the basic
scheme of the algorithm is the following:

1. Each process stores in an Actives set its own ID and the IDs of processes
   from which it recently receives ALIVE.
2. Each process periodically chooses its leader in its Actives set.

Due to the arbitrary initial configuration, variables may have initial strange
values. To deal with this, variables are either only incremented or periodi-
cally refreshed. Variables as $\text{Counter}_p[p]$ can be only incremented. Variables
as $\text{Counter}_p[q]$ is refreshed each time a message is received from $q$. Variables as
Collect$_p$ are refreshed at each complete loop iteration by what has happened
in the current loop and in the previous one.

Hence, after a constant number of iteration loop and the reception of messages
effectively sent by processes, the value of these variables become “correct”. For
example, $\text{Counter}_p[q]$ will be the value known by process $p$ of $\text{Counter}_q[q]$. Mor-
Moreover, we have the property that: $\text{Counter}_q[q]$ is bounded by $c$ if and only
if $\text{Counter}_p[q]$ is bounded by $c$.

As in [14], several problems have to be solved. The first one concerns the way
a process chooses its leader. A simple way to choose a leader is to choose
the smallest identifier in Actives. However, due to asynchrony of the links,
the leadership can oscillate among some alive processes. To fix this problem,
Aguilera et al propose in [14] the use of accusation’s counters: each process
$p$ stores in $\text{Counter}_p[p]$ an estimation of how many times it was previously
suspected to be crashed by all other processes. When $p$ sends an ALIVE mes-
\begin{itemize}
  \item \begin{itemize}
    \item $\text{Counter}_p[p]$ is refreshed each time a message is received from $q$.
    \item Collect$_p$ are refreshed at each complete loop iteration.
  \end{itemize}
\end{itemize}
Hence, after a constant number of iteration loop and the reception of messages
effectively sent by processes, the value of these variables become “correct”. For
example, $\text{Counter}_p[q]$ will be the value known by process $p$ of $\text{Counter}_q[q]$. Mor-
Moreover, we have the property that: $\text{Counter}_q[q]$ is bounded by $c$ if and only
if $\text{Counter}_p[q]$ is bounded by $c$.

As in [14], several problems have to be solved. The first one concerns the way
a process chooses its leader. A simple way to choose a leader is to choose
the smallest identifier in Actives. However, due to asynchrony of the links,
the leadership can oscillate among some alive processes. To fix this problem,
Aguilera et al propose in [14] the use of accusation’s counters: each process
$p$ stores in $\text{Counter}_p[p]$ an estimation of how many times it was previously
suspected to be crashed by all other processes. When $p$ sends an ALIVE mes-
\begin{itemize}
  \item \begin{itemize}
    \item $\text{Counter}_p[p]$ is refreshed each time a message is received from $q$.
    \item Collect$_p$ are refreshed at each complete loop iteration.
  \end{itemize}
\end{itemize}
Hence, after a constant number of iteration loop and the reception of messages
effectively sent by processes, the value of these variables become “correct”. For
example, $\text{Counter}_p[q]$ will be the value known by process $p$ of $\text{Counter}_q[q]$. Mor-
Moreover, we have the property that: $\text{Counter}_q[q]$ is bounded by $c$ if and only
if $\text{Counter}_p[q]$ is bounded by $c$.

As in [14], several problems have to be solved. The first one concerns the way
a process chooses its leader. A simple way to choose a leader is to choose
the smallest identifier in Actives. However, due to asynchrony of the links,
the leadership can oscillate among some alive processes. To fix this problem,
Aguilera et al propose in [14] the use of accusation’s counters: each process
$p$ stores in $\text{Counter}_p[p]$ an estimation of how many times it was previously
suspected to be crashed by all other processes. When $p$ sends an ALIVE mes-
\begin{itemize}
  \item \begin{itemize}
    \item $\text{Counter}_p[p]$ is refreshed each time a message is received from $q$.
    \item Collect$_p$ are refreshed at each complete loop iteration.
  \end{itemize}
\end{itemize}
Hence, after a constant number of iteration loop and the reception of messages
effectively sent by processes, the value of these variables become “correct”. For
example, $\text{Counter}_p[q]$ will be the value known by process $p$ of $\text{Counter}_q[q]$. Mor-
Moreover, we have the property that: $\text{Counter}_q[q]$ is bounded by $c$ if and only
if $\text{Counter}_p[q]$ is bounded by $c$.
Algorithm 3 CE-PSSCF Leader Election on $S_2$, code for every process $p$

1: variables:
2: $\text{Leader}_p \in \{1, \ldots, n\}$, $\text{OldLeader}_p \in \{1, \ldots, n\}$
3: $\text{SendTimer}_p$, $\text{ReceiveTimer}_p$: non-negative integers
4: $\text{Counter}_p[1 \ldots n]$, $\text{Phase}_p[1 \ldots n]$: arrays of non-negative integers
5: $\text{Collect}_p$, $\text{OtherActives}_p$: sets of non-negative integers
6: $\text{CheckCollect}_p$, $\text{CheckSet}_p$: sets of 2-uple of integers
7: macro:
8: $\text{Actives}_p = \text{OtherActives}_p \cup \{p\}$
9: repeat forever
10: for all $q \in V \setminus \{p\}$ do
11: if receive($\text{ALIVE}_q.qnt.qph \rangle$) from $q$ then
12: $\text{Collect}_p = \text{Collect}_p \cup \{q\}$
13: $\text{Counter}_p[q] = qnt$
14: $\text{Phase}_p[q] = qph$
15: if $q \neq \text{Leader}_p$ then /* $q$ is not the leader of $p$, so */
16: send($\text{CHECK}. \text{Leader}_p.\text{Phase}_p[p] \rangle$) to $q$ /* $p$ asks $q$ to check its leader */
17: end if
18: end if
19: if receive($\text{ACCUSE}_r.rph \rangle$) from $q$ then
20: if $r = p$ then
21: if $rph = \text{Phase}_p[p]$ then /* $p$ tests if the accusation is legitimate */
22: $\text{Counter}_p[p] = \text{Counter}_p[p] + 1$
23: end if
24: else
25: send($\text{ACCUSE}_p-r.rph \rangle$) to $r$ /* $p$ relays the accusation */
26: end if
27: end if
28: if receive($\text{CHECK}.r.rph \rangle$) from $q$ then
29: $\text{CheckCollect}_p = \text{CheckCollect}_p \cup \{(r,rph)\}$
30: end if
31: end for
32: $\text{SendTimer}_p = \text{SendTimer}_p + 1$
33: if $\text{SendTimer}_p \leq \sigma$ then
34: if $\text{Leader}_p = p$ then /* if $p$ believes to be the leader, then */
35: send($\text{ALIVE}. \text{Counter}_p[p].\text{Phase}_p[p] \rangle$) to every process except $p$ /* $p$ sends ALIVE */
36: end if
37: $\text{SendTimer}_p = 0$
38: end if
39: $\text{ReceiveTimer}_p = \text{ReceiveTimer}_p + 1$
40: if $\text{ReceiveTimer}_p > 5\delta$ then
41: for all $q \in \text{OtherActives}_p \setminus (\text{Collect}_p \cap \text{OtherActives}_p)$ do
42: send($\text{ACCUSE}_q.q.qph \rangle$) to every process except $p$
43: end for
44: for all $(r,rph) \in \{(q,qph) \in \text{CheckSet}_p, q \notin \text{Collect}_p\}$ do
45: send($\text{ACCUSE}_p-r.rph \rangle$) to every process except $p$
46: end for
47: $\text{OtherActives}_p = \text{Collect}_p$
48: $\text{Collect}_p = \emptyset$
49: $\text{CheckSet}_p = \text{CheckCollect}_p$
50: $\text{CheckCollect}_p = \emptyset$
51: $\text{OldLeader}_p = \text{Leader}_p$
52: $\text{Leader}_p \leftarrow r$ such that $(\text{Counter}_p[r],r) = \min\{(\text{Counter}_p[q],q) : q \in \text{Actives}_p\}$
53: if $(\text{OldLeader}_p = p) \land (\text{Leader}_p \neq p)$ then /* $p$ looses its leadership */
54: $\text{Phase}_p[p] = \text{Phase}_p[p] + 1$
55: end if
56: $\text{ReceiveTimer}_p = 0$
57: end if
58: end repeat

remains bounded and, so, a source is eventually elected.

As, several links are not fair in $S_2$, several ACCUSE-$r$ messages can be lost. To solve this problem, each ACCUSE-$r$ message is relayed once. If a process $q$ accuses a process $p$ infinitely often, with the help of the fair hub, $p$ receives
infinitely many ACCUSE-$p$ messages. Note that this scheme preserves the communication efficiency: once a permanent leader is elected, there is no new accusation and the relaying stops.

The second problem is the following. The aim of the accusation counter is that the processes that communicate well do not increase their accusation counter infinitely many times.

A source $s$ can stop to consider itself as the leader when it selects another process $p$ as its leader (a process in $Actives_s$ with a smaller counter). In this case, the source voluntarily stops sending ALIVE messages for the communication efficiency. To avoid that other process that considered $s$ as its leader eventually suspects $s$ and sends ACCUSE-$s$ messages a mechanism is added so that a source increments its own accusation counter only a finite number of times. A process now increments its accusation counter only if it receives a "legitimate" accusation: an accusation due to the delay or the loss of one of its ALIVE message and not due to the fact that it voluntarily stopped sending messages. To detect if an accusation is legitimate, each process $p$ saves in $Phase_p[p]$ the number of times it looses the leadership in the past and includes this value in each of its ALIVE messages. Hence, when $q$ wants to accuse $p$, it now includes its own view of $p$’s phase number in the ACCUSE-$p$ message. This ACCUSE-$p$ message will be considered as legitimate by $p$ only if the phase number it contains matches the current phase value of $p$. Also, whenever $p$ loses the leadership and stops sending ALIVE messages voluntarily, $p$ increments $Phase_p[p]$ and does not send the new value to any other process.

There is a last problem to solve. Due to the fact that several links can be lossy, the alive processes may be split into two subsets $\Pi_p$ and $\Pi_q$ such that processes in $\Pi_p$ (including $p$) and processes in $\Pi_q$ (including $q$) have $p$ and $q$ as permanent leader, respectively. To prevent this problem while preserving the communication-efficiency, we use the fact that the fair hub $h$ receives timely ALIVE messages from both $p$ and $q$. When $h$ receives an ALIVE message from a process $q$ that is not its leader, it sends a (CHECK,$p$,php) message to $q$ where php corresponds to the phase value of its leader $p$. When $q$ receives such a message it stores the tuple ($p$,php) into a list and waits for receiving an ALIVE message from $p$. If the link ($p,q$) is too slow or lossy, $q$ eventually sends an (ACCUSE-$p$,php) message. Hence, using this method, the previous problem is solved: $p$ eventually loses its leadership due to the ACCUSE-$p$ messages generated by $q$.

5.3 Correctness of Algorithm 3 in $S_2$

We recall that in the proofs, we assume that the initial configuration of the system is arbitrary. Also, we will denote by $var_p^t$ the value of $var_p$ at time $t$. Finally, we will denote by $s$ the timely source and by $h$ the fair hub.
Lemma 12 Let \( p \) and \( q \) be two distinct alive processes. If \( q \in \text{Actives}_p \) holds infinitely often, then \( p \) receives ALIVE messages from \( q \) infinitely often.

Proof. As \( q \neq p \), \( q \in \text{Actives}_p \) holds infinitely often and implies that \( q \in \text{Collect}_p \) holds infinitely often. Now, \( \text{Collect}_p \) is periodically reset to \( \emptyset \) and \( q \) is inserted into \( \text{Collect}_p \) only if \( p \) receives ALIVE from \( q \), hence the lemma holds.

According to Remark 1 (page 8), we have the following observation:

Observation 7 After the first loop iteration (i.e., after at most \( \beta \) times), \( \text{Counter}_p[p] \) and \( \text{Phase}_p[p] \) are monotonically nondecreasing with time.

Lemma 13 Let \( p \) and \( q \) be two distinct processes. If \( p \) receives ALIVE messages from \( q \) infinitely often, then (1) \( q \) is alive and (2) for every time \( t > \beta \), there is a time after which \( \text{Counter}_p[q] \geq \text{Counter}^t_q[q] \) and \( \text{Phase}_p[q] \geq \text{Phase}^t_q[q] \) forever.

Proof Outline. Point (1) is straightforward because a crashed process stops sending messages. Point (2) is due to Observation 7 and the fact that \( q \) always stores in \( \text{Counter}^t_q[q] \) and \( \text{Phase}_q[q] \) the most up-to-date values of \( \text{Counter}_p[q] \) and \( \text{Phase}_p[q] \) it receives from \( p \).

By Lemmas 12, 13 and Observation 7, follows:

Lemma 14 Let \( p \) and \( q \) be two alive processes. If \( q \in \text{Actives}_p \) holds infinitely often, then \( q \) is alive and, for every time \( t > \beta \), there is a time after which \( \text{Counter}_p[q] \geq \text{Counter}^t_q[q] \) and \( \text{Phase}_p[q] \geq \text{Phase}^t_q[q] \) forever.

As all input and output links of \( h \) are fair, we have:

Lemma 15 For every alive process \( p \neq h \), (1) if \( p \) sends a message of type \( T \) to \( h \) infinitely often, then \( h \) receives a message of type \( T \) from \( p \) infinitely often, and (2) if \( h \) sends a message of type \( T \) to \( p \) infinitely often, then \( p \) receives a message of type \( T \) from \( h \) infinitely often.

According to Remark 1 (page 8), we have the two following observations:

Observation 8 A process \( p \) sends ALIVE at time \( t > \beta \) only if \( \text{Leader}_p = p \) at time \( t \).

Observation 9 A process \( p \) switches \( \text{Leader}_p \) from \( p \) to \( q \neq p \) at time \( t > \beta \) only if \( \text{OldLeader}_p = p \) at time \( t \).

Lemma 16 There exists a time \( t > \beta \) such that, for every process \( p \neq s \) and every \( k \geq 0 \), if \( s \) sends (ALIVE, -, k) to \( p \) at time \( t' > t \), then:
1. \( s \) sends another (ALIVE, -, k) message to \( p \) during the time interval \( ]t', t' + \)
δ + β], or
2. Phase\(s[s] > k\) holds forever from time \(t' + δ + β\).

**Proof.** There is a time \(t_0 > β\) after which no message initially in the links is still in transit. Assume that \(s\) sends (ALIVE,−,k) to \(p\) at time \(t_1 > t_0\). Then, \(Leader_s = s\) (Observation 8) and \(Phase_s[s] = k\) at time \(t_1\). There are two cases:

- \(s\) switches \(Leader_s\) from \(s\) to \(q \neq s\) at time \(t_2 \in [t_1, t_1 + δ]\).

\(OldLeader_s = s\) holds at time \(t_2\) (Observation 9) and \(s\) increments \(Phase_s[s]\) before time \(t_2 + β\). By Observation 7, \(Phase_s[s]\) becomes strictly greater than \(k\) forever before time \(t_2 + β\), which proves the lemma in this case.

- \(s\) continuously satisfies \(Leader_s = s\) during the time interval \([t_1, t_1 + δ]\).

First, as \(Leader_s = s\) during \([t_1, t_1 + δ]\), \(Phase_s[s]\) stays equal to \(k\) during this interval and it remains to prove that \(s\) ALIVE during the interval.

After \(s\) sends (ALIVE,−,k) to \(p\) at time \(t_1\), \(s\) resets \(SendTimer_s\) to 0 in the same loop iteration. So, when the program counter points to the first instruction of the loop at time \(t_2 \in [t_1, t_1 + β]\), \(SendTimer_s = 0\). From time \(t_2\), \(s\) executes a complete loop iteration at least every \(β\) times. After executing \(σ - 1\) iterations, the program counter points to the first instruction at time \(t_3 \leq t_1 + δ\), \(SendTimer_s = σ - 1\), and \(Leader_s\) is equal to \(s\). Consider now the next loop iteration: \(s\) increments \(SendTimer_s\) to \(σ\) (Line 32) and satisfies the tests of Lines 33-34 and sends another ALIVE message in Line 35, i.e., at time \(t_4 \in [t_3, t_1 + δ + β]\), which proves the lemma.

\[\square\]

**Lemma 17** There exists a time \(t > β\) such that, for all distinct processes \(p\) and \(r\) and every \(k \geq 0\), if \(p\) receives a (CHECK\(r\),r,k) message at some time \(t' \geq t\), then \(r\) sent an (ALIVE,−,k) message before time \(t'\).

**Proof Outline.** The lemma is straightforward if \(p\) receives only a finite number of (CHECK\(r\),r,−) messages. So, assume that \(p\) receives infinitely many (CHECK\(r\),r,−) messages. Then, there is a process \(q\) that sends infinitely many (CHECK\(r\),r,−) messages to \(p\). In this case, \(r \in Actives_q\) holds infinitely often. By Lemma 12, \(q\) receives ALIVE messages from \(r\) infinitely often. This means that \(r\) sends ALIVE messages to \(q\) infinitely often. Now, \(q\) keeps in \(Phase_q[r]\) the most recent phase value received from \(r\). So, there is a time after which if \(q\) sends (CHECK\(r\),r,k), then \(r\) previously sent (ALIVE,−,k).

\[\square\]

When a message \(m\) is delivered to a process \(p\), \(p\) receives a message of the same type of \(m\) at most one complete loop iteration after the delivery of \(m\). Hence, by Observation 5 (page 7) and as \(s\) is a timely source:

**Lemma 18** 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 \([t, t + δ + 2β]\).
Lemma 19  In any execution, $\text{Counter}_s[s]$ is bounded.

**Proof.** Assume, by contradiction, that $\text{Counter}_s[s]$ is unbounded. So, there is a process $p \neq s$ that sends $\text{ACCUSE}-s$ messages infinitely often. There is three possibilities for a process $p$ to send an $\text{ACCUSE}-s$ message: either $p$ executes Line 25, Line 42, or Line 45. Line 25 is used to relay $\text{ACCUSE}-s$ messages. So, without the loss of generality, we can assume that $p$ executes Line 42 or Line 45 infinitely often:

(a) $p$ accuses $s$ infinitely often in Line 42.

$p$ executes at least $5\delta + 1$ loop iterations between each attempt of accusation in Lines 41-43, i.e., at least $5\delta + 1$ times. So, $p$ accuses $s$ infinitely often in Line 42 only if the following scheme arrives infinitely often:

- $s$ is inserted into $\text{Collect}_p$ after $p$ receives an $\text{ALIVE}$ message from $s$.
- Then, $\text{OtherActives}_p$ is set to $\text{Collect}_p$ and $\text{Collect}_p$ is set to $\emptyset$.
- Finally, during the following period of $5\delta + 1$ times, $s$ is not inserted into $\text{Collect}_p$ meaning that $p$ does not received any $\text{ALIVE}$ messages from $s$ during the period.

Hence, $p$ must receive $\text{ALIVE}$ messages from $s$ infinitely often in such a way that infinitely often the time between two receptions of $\text{ALIVE}$ messages is greater than $5\delta + 1$ times. Now, by Lemmas 16 and 18, there is a time from which when $p$ receives an $(\text{ALIVE},-,k)$ message from $s$ at time $t$, either $p$ receives another $(\text{ALIVE},-,k)$ message from $s$ during the time interval $[t,t+2\delta+3]$ or $\text{Phase}_s[s] > k$ holds forever from time $t+\delta+1$. In the former case, $p$ does not accuse $s$. In the latter case, $p$ sends an $(\text{ACCUSE}-s,k)$ message to all other processes after time $t+5\delta+1$. Now, even if $s$ eventually received all these accusations, these accusations do not cause any incrementation of $\text{Counter}_s[s]$ because $\text{Phase}_s[s] > k$ when $s$ received them. Hence, the contradiction.

(b) $p$ accuses $s$ infinitely often in Line 45. Using the arguments similarly to those in Case (a), we can deduce that $p$ must receive $(\text{CHECK},s,-)$ messages infinitely often and Lemma 17 implies that $s$ sends $(\text{ALIVE},-,k)$ messages to $p$ infinitely often. As the link $(s,p)$ is timely, $p$ receives $(\text{ALIVE},-,k)$ messages from $s$ infinitely often. Similar to (a) again, $p$ executes Line 45 infinitely often only if the following situation arrives infinitely often: the time between the reception of a $(\text{CHECK},s,-)$ message and the next $\text{ALIVE}$ message from $s$ is greater than $5\delta + 1$ times and similarly to the previous case, we obtain a contradiction.

$\square$

**Definition 1** For each process $p$, let $c_p$ be the largest value of $\text{Counter}_p[p]$ in the execution that we consider ($c_p = \infty$ if $\text{Counter}_p[p]$ is unbounded). Let $\ell$ be the process such that $(c_{\ell},\ell) = \min\{(c_p,p); p \text{ is an alive process}\}$.

23
By Definition, $\ell$ is an alive process. Furthermore, $c_s < \infty$ by Lemma 19, so, $c_\ell < \infty$; i.e., $Counter_\ell[\ell]$ is bounded.

The following lemma is straightforward by definition of $\ell$ and by the way $p$ computes $Leader_p$.

**Lemma 20** For every alive process $p$, if there is a time after which $\ell \in Actives_p$ forever, then there is a time after which $Leader_p = \ell$ forever.

**Corollary 3** There is a time after which $Leader_\ell = \ell$ forever.

**Lemma 21** There is a time after which $Phase_\ell[\ell]$ stops changing.

**Proof.** $\ell$ changes $Phase_\ell[\ell]$ infinitely often only if $\ell$ switches $Leader_\ell$ from $\ell$ to a process $q \neq \ell$ infinitely often, which contradicts Corollary 3. $\square$

**Definition 2** Let $\ellphase$ be the final value of $Phase_\ell[\ell]$.

**Lemma 22** For every alive process $p$, there is a time after which if $Leader_p = \ell$ infinitely often, then $Phase_p[\ell] \geq \ellphase$.

**Proof.** Let $p$ be any process. Assume that eventually $Leader_p = \ell$ holds infinitely often. If $p = \ell$, then the lemma trivially holds by the definition of $\ellphase$. If $p \neq \ell$, then, $\ell \in Actives_p$ holds infinitely often. By Lemma 14, there is a time from which $Phase_p[\ell] \geq \ellphase$ forever, which proves the lemma. $\square$

**Lemma 23** Any process $p$ can send only a finite number of $(\text{ACCUSE}-\ell,k)$ messages with $k < \ellphase$.

**Proof.** First, as $Phase_\ell[\ell] = \ellphase$ eventually holds forever, (1) $\ell$ can only send a finite number of $(\text{ALIVE},-,k)$ messages with $k < \ellphase$. We now show, by contradiction, that (2) only a finite number of $(\text{CHECK},\ell,k)$ messages are sent with $k < \ellphase$.

Assume that infinitely many $(\text{CHECK},\ell,k)$ messages are sent with $k < \ellphase$. Then, $Leader_q = \ell$ holds infinitely often and by Lemma 22 there is a time after which $Phase_q[\ell] \geq \ellphase$ forever. From this time $q$ no $(\text{CHECK},\ell,k)$ message forever, a contradiction.

We now show, by contradiction, that a process $p$ sends $(\text{ACCUSE}-\ell,k)$ messages with $k < \ellphase$ in Lines 42 or 45 only finitely many time. This claim immediately implies the lemma because eventually a process can relay an ACCUSE-$\ell$ message in Line 25 only if another process previously sends this message in Lines 42 or 45. Assume, by contradiction, that $p$ sends infinitely many $(\text{ACCUSE}-\ell,k)$ messages with $k < \ellphase$ in lines 42 or 45. Then, (a) $\ell \in OtherActives_p$ or (b) $(\ell,k) \in CheckSet_p$ with $k < \ellphase$ holds infinitely often. In Case (a), by Lemma 14 that there is a time from
which \( \text{Phase}_p[\ell] = \ell_{\text{phase}} \) holds forever and, from this time, \( p \) never more sends any \((\text{ACCUSE}-\ell,k)\) with \( k < \ell_{\text{phase}} \). In Case (b), it is easy to see that \((\ell,k) \in \text{CheckSet}_p\) with \( k < \ell_{\text{phase}} \) holds infinitely often only if \( q \) receives \((\text{CHECK},\ell,k)\) messages infinitely often, a contradiction to Claim (2).

Lemma 24. No process sends \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) messages infinitely often in Lines 42 or 45.

Proof Outline. Assume that a process \( p \) sends \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) messages infinitely often in Lines 42 or 45. If \( p = h \), then \( \ell \) receives infinitely many \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) by Lemma 15. Otherwise, \( h \) relays infinitely many \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) to \( \ell \) and, so, \( \ell \) receives infinitely many \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) by Lemma 15. Hence, in any case, \( \ell \) receives infinitely many \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) which is a contradiction because eventually each time \( \ell \) receives such a message, \( \ell \) increments \( \text{Counter}_\ell[\ell] \) and so eventually \( \text{Counter}_\ell[\ell] \) becomes greater than \( c_\ell \).

Lemma 25. No process \( p \) adds and removes \( \ell \) to and from \( \text{Active}_p \) infinitely often.

Proof. Assume, by contradiction, that some process \( p \) adds and removes \( \ell \) to and from \( \text{Active}_p \) infinitely often. This implies that (a) \( p \) receives \((\text{ALIVE},-,-)\) messages from \( \ell \) infinitely often, and (b), \( p \) sends \((\text{ACCUSE}-\ell,-)\) messages infinitely often in Line 42. From (a), the definition of \( \ell_{\text{phase}} \), and the fact that the link \((\ell,p)\) initially contains only a finite number of messages, we can deduce that eventually \( p \) only receives \( \text{ALIVE} \) messages from \( \ell \) of the form \((\text{ALIVE},-,-,\ell_{\text{phase}})\). So, there is a time after which \( \text{Phase}_p[\ell] = \ell_{\text{phase}} \) forever. Thus, from (b), \( p \) sends infinitely many \((\text{ACCUSE}-\ell,\ell_{\text{phase}})\) messages in Line 42 — a contradiction to Lemma 24.

Lemma 26. There is a time after which \( \ell \in \text{Active}_h \) and \( \text{Phase}_h[\ell] = \ell_{\text{phase}} \) holds forever.

Proof. If \( h = \ell \), then the result follows by the definition of \( \ell_{\text{phase}} \) and the fact that \( \ell \in \text{Active}_\ell \) always holds. Consider now that \( h \neq \ell \). By Corollary 3 and the definition of \( \ell_{\text{phase}} \), \( \ell \) sends an infinite number of \((\text{ALIVE},-,-,\ell_{\text{phase}})\) messages to all processes except itself. Moreover, \( \ell \) only sends a finite number of \((\text{ALIVE},-,-,y)\) messages with \( y \neq \ell_{\text{phase}} \). Since \( h \neq \ell \), \( h \) receives an infinite number of these \((\text{ALIVE},-,-,\ell_{\text{phase}})\) messages from \( \ell \) by Lemma 15. Therefore, there is a time after which \( h \) satisfies \( \text{Phase}_h[\ell] = \ell_{\text{phase}} \) forever. Moreover, \( \ell \in \text{Active}_h \) holds infinitely often. From Lemma 25, \( h \) removes \( \ell \) from \( \text{Active}_h \) only finitely often, and the lemma holds.

By Lemmas 20 and 26, we can deduce the following lemma:
Lemma 27 There is a time after which Leader$_h = \ell$ holds forever.

Lemma 28 There is a time after which only $\ell$ sends ALIVE messages.

Proof. Consider any alive process $p \neq \ell$. From Lemma 25, there are two possible cases:

1. There is a time after which $\ell \in \text{Actives}_p$ holds forever. In this case, there is a time after which Leader$_p = \ell$ by Lemma 20. After this time, $p$ does not send ALIVE messages.
2. There is a time after which $\ell \notin \text{Actives}_p$ holds forever. This implies that:
   (a) There is a time after which $p$ receives no ALIVE message from $\ell$ forever.
   (b) $p \neq h$ by Lemma 26.
   (c) $h \neq \ell$ (because if $h = \ell$, then, by Corollary 3, $h$ sends an infinite number of ALIVE messages to $p$, and, by Lemma 15, $p$ receives an infinite number of ALIVE messages from $h$, which contradicts (a)).

Assume now, by contradiction, that $p$ sends ALIVE messages infinitely often. By Lemma 15, $h$ receives ALIVE messages from $p$ infinitely often. By Lemmas 26 and 27, there is a time after which Leader$_h = \ell$ and Phase$_h[\ell] = \ell$phase forever. After that time, each time $h$ receives an ALIVE message from $p$, $h$ sends a (CHECK,$\ell$,$\ell$phase) message to $p$ (since $p \neq \ell$ and Leader$_h = \ell$). Hence, $h$ sends infinitely many (CHECK,$\ell$,$\ell$phase) messages to $p$ and there is a time after which $p$ only receives (CHECK,$\ell$,$\ell$phase) messages from $h$. By Lemma 15, $p$ receives such messages infinitely often. Hence, ($\ell$,$\ell$phase) is inserted infinitely often in CheckCollect$_p$ and, as a consequence, in CheckSet$_p$. Now, there is a time after which $\ell \notin \text{Actives}_p$ holds forever and, as a consequence, $\ell \notin \text{Collect}_p$ forever from this time. Hence, $p$ sends (ACCUSE-$\ell$,$\ell$phase) messages in Line 45 infinitely often — a contradiction to Lemma 24.

Hence, in both Cases (1) and (2), there is a time after which $p$ no more sends any ALIVE message.

Lemma 29 There is a time after which every process $p$ satisfies Leader$_p = \ell$ forever.

Proof. Let $p$ be any alive process. By Lemma 25, there are two possible cases:

1. There is a time after which $\ell \in \text{Actives}_p$ holds forever. In this case, by Lemma 20, there is a time after which Leader$_p = \ell$ forever.
2. There is a time after which $\ell \notin \text{Actives}_p$ holds forever. Since a process $q \neq p$ can remain in Active$_p$ only if $p$ keeps receiving ALIVE messages from $q$, then, by Lemma 28 and the fact that $p \in \text{Active}_p$ (always), there is a time after which Actives$_p = \{p\}$ holds forever. This implies that Leader$_p = p$ eventually holds forever and, as a consequence, $p$ repeatedly sends ALIVE message forever — a contradiction to Lemma 28.

Thus, only Case (1) holds.

26
Lemma 30 There is a time after which only $\ell$ sends messages.

Proof. There are three types of messages:

(a) By Lemma 28, there is a time after which only $\ell$ sends ALIVE messages.
(b) Only a finite number of CHECK messages are sent. By Lemma 29, eventually every process $p$ satisfies $Leader_p = \ell$. By (a), every process $p$ eventually only receives ALIVE messages from $\ell$. Since these two conditions hold, no CHECK message is sent forever.
(c) For every process $q$, only a finite number of ACCUSE-$q$ messages are sent. To see this we now show that any process $p$ can only send finitely many ACCUSE-$q$ messages in Lines 42 or 45. As Line 25 is only used to relay the ACCUSE-$q$ messages, this implies the claim. So, assume, by contradiction, that some process $p$ sends infinitely many ACCUSE-$q$ messages in Lines 42 or 45. First, because of Case (b), $p$ cannot send (ACCUSE-$q$,$-$) messages infinitely often in Line 45. Then, $p$ sends (ACCUSE-$q$,$-$) messages infinitely often in Line 42 and, as a consequence, $p$ receives infinitely many ALIVE messages from $q$. By Lemma 28 and owing to the fact that there is only a finite number of messages initially in the links, any process $p$ can only send ACCUSE-$q$ messages infinitely often if $q = \ell$ by Lemma 28. As the number of messages initially in the links is finite and eventually $\ell$ only sends (ALIVE,$-$,$\ell$phase) messages, eventually $p$ only receives ALIVE message from $\ell$ of the form (ALIVE,$-$,$\ell$phase) and, as a consequence, $Phase_p[\ell] = \ell$phase eventually holds forever. Hence, eventually $p$ only sends ACCUSE-$\ell$ message to $\ell$ of the form: (ACCUSE-$\ell$,$\ell$phase) which contradicts Lemma 24.

By Lemmas 29 and 30, we obtain the following theorem:

Theorem 7 Algorithm 3 implements a CE-PSSCF leader election in $S_2$.

6 System $S_1$

Contrary to $S_2$, the existence of a fair hub is not required in $S_1$. This difference may seem to be weak but actually is important. Indeed, contrary to $S_2$, CE-PSSCF is not possible in $S_1$. To see this: let $S_1^-$ as being any system $S_0$ with an eventual timely source and $n \geq 3$ processes. In [14], Aguilera et al show that there is no communication-efficient fault-tolerant leader election algorithm in $S_1^-$. Now, any PSSCF leader election algorithm in $S_1$ is also a PSSCF leader election algorithm in $S_1^-$ by Theorem 2 (page 9). Hence:

Theorem 8 There is no CE-PSSCF leader election algorithm in $S_1$ with $n \geq 3$ processes.
We now show that PSSCF leader election can be non-communication-efficiently implemented in $S_1$. To that goal, we adapted the fault-tolerant but non-communication-efficient algorithm for $S_1$ given in [14].

6.1 PSSCF Leader Election in $S_1$

Algorithm 4 implements a PSSCF leader election in $S_1$. It resumes the principle of accusation’s counter presented in the previous section but in a simpler version: because we do not implement the communication-efficiency, every process now periodically sends ALIVE messages to each other. Each process $p$ still keeps track in $Counter_p[q]$ of how many times process $q$ (including $p$) was previously suspected of being crashed. Finally, $p$ computes an $Actives_p$ set: this set still corresponds to its own ID plus the IDs of the processes from which it recently receives ALIVE (i.e., the processes it trusts to be alive).

However, the way $p$ computes its leader is quite different. First, $p$ periodically evaluates a “local” leader. Then, $p$ periodically chooses a “global” leader among the local leaders of the processes in its $Actives$ set.

The local leader of $p$ is stored in $LLeader_p[p]$ and corresponds to the process of $Actives_p$ in with the fewest number of suspicions (we use the IDs to break ties). Then, $p$ has to evaluate the local leaders of all the processes in $Actives_p$. To that goal, every process $q$ now includes the ID of its current local leader in its ALIVE messages. Thanks to the ALIVE messages, $p$ can now maintain in $LLeader_p[q]$ the local leader of each process $q$ in $Actives_p$. It remains then to periodically select a “global” leader for $p$ in the set $\{LLeader_p[q] : q \in Actives_p\}$.

To select its global leader, any process $p$ maintains another array of accusation counters: $LLCounter_p$. In $LLCounter_p[q]$, $p$ stores the $q$’estimation of how many times $q$’own local leader was previously suspected of having crashed, i.e., the most up-to-date value of $Counter_q[LLeader_q[q]]$. To that goal, the value $Counter_q[LLeader_q[q]]$ is also included in all the ALIVE messages sent by $q$. Hence, $p$ can now periodically select its global leader as the process $\ell$ with the smallest $(LLCounter_p[\ell],\ell)$ tuple in the set $\{LLeader[q] : q \in Actives_p\}$.

We deal with the fact that the initial configuration is arbitrary as in the previous algorithm.

6.2 Correctness of Algorithm 4 in $S_1$

We recall that we assume that the initial configuration is arbitrary. Also, we will denote by $var^t_p$ the value of $var_p$ at time $t$ and by $s$ the timely source.

The proof of the next lemma is identical to the one of Lemma 12, page 20.
Algorithm 4 PSSCF Leader Election on $\mathcal{S}_1$, code for each process $p$

1: variables:
2: $\text{Leader}_p \in \{1, \ldots, n\}$
3: $\text{SendTimer}_p, \text{ReceiveTimer}_p$: non-negative integers
4: $\text{LLeader}_p[1 \ldots n], \text{LLCounter}_p[1 \ldots n], \text{Counter}_p[1 \ldots n]$; arrays of non-negative integers
5: $\text{Collect}_p, \text{OtherActives}_p$: sets of non-negative integers
6: macros:
7: $\text{Actives}_p = \text{OtherActives}_p \cup \{p\}$
8: $\text{MyLLeader}_p = r, (\text{Counter}_p[r], r) = \min \{(\text{Counter}_p[q], q) : q \in \text{Actives}_p\}$
9: $\text{MyLeader}_p = \ell, (\text{LLCounter}_p[\ell], \text{LLeader}_p[\ell]) = \min \{(\text{LLCounter}_p[q], \text{LLeader}_p[q]) : q \in \text{Actives}_p\}$
10: repeat forever
11: for all $q \in V \setminus \{p\}$ do
12: if receive(ACCUSE) from $q$ then
13: $\text{Counter}_p[p] \leftarrow \text{Counter}_p[p] + 1$
14: end if
15: if receive(ALIVE, $r, rcnt, qcnt$) from $q$ then
16: $\text{Collect}_p \leftarrow \text{Collect}_p \cup \{q\}$
17: $\text{Counter}_p[q] \leftarrow qcnt$
18: $\text{LLeader}_p[q] \leftarrow r$
19: $\text{LLCounter}_p[q] \leftarrow rcnt$
20: end if
21: end for
22: $\text{SendTimer}_p \leftarrow \text{SendTimer}_p + 1$
23: if $\text{SendTimer}_p \geq \sigma$ then
24: send(ALIVE, $\text{LLeader}_p[p], \text{Counter}_p[\text{LLeader}_p[p]], \text{Counter}_p[p]$) to every process except $p$
25: $\text{SendTimer}_p \leftarrow 0$
26: end if
27: $\text{ReceiveTimer}_p \leftarrow \text{ReceiveTimer}_p + 1$
28: if $\text{ReceiveTimer}_p > 5\delta$ then
29: $\text{OtherActives}_p \leftarrow \text{Collect}_p$
30: for all $q \in V \setminus \text{Actives}_p$ do
31: send(ACCUSE) to $q$
32: end for
33: $\text{LLeader}_p[p] \leftarrow \text{MyLLeader}_p$
34: $\text{LLCounter}_p[p] \leftarrow \text{Counter}_p[\text{LLeader}_p[p]]$
35: $\text{Leader}_p \leftarrow \text{MyLeader}_p$
36: $\text{Collect}_p \leftarrow \emptyset$
37: $\text{ReceiveTimer}_p \leftarrow 0$
38: end if
39: end repeat

Lemma 31 For every alive process $p$ and every process $q \neq p$, if $q \in \text{Actives}_p$ holds infinitely often, then $p$ receives ALIVE messages from $q$ infinitely often.

Observation 10 $\text{Counter}_p[p]$ is monotonically nondecreasing with time from time $\beta$.

The proof of the next lemma is similar to the one of Lemma 13, page 21.

Lemma 32 Let $p$ and $q$ be two distinct processes. If $p$ receives ALIVE messages from $q$ infinitely often, then $q$ is alive and, for every time $t > \beta$, there is a time after which $\text{Counter}_p[q] \geq \text{Counter}_q^t[q]$ forever.

The proof of the next lemma is similar to the one of Lemma 14, page 21.

Lemma 33 For every alive process $p$ and every process $q$, if $q \in \text{Actives}_p$ holds infinitely often, then $q$ is alive and, for every time $t$, there is a time after which $\text{Counter}_p[q] \geq \text{Counter}_q^t[q]$ forever.
Lemma 34  $s$ sends an ALIVE message to $p$ at most every $\delta + \beta$ times.

Proof.  Consider any time $t$. At time $t$, $\text{SendTimer}_s \geq 0$. From time $t$, the program counter of $s$ points to the first instruction of the loop at time $t'$ such that $t < t' \leq t + \beta$. From time $t'$, $s$ increments $\text{SendTimer}_s$ once by loop iteration, i.e., at most every $\beta$ times. So, the test $\text{SendTimer}_s \geq \sigma$ becomes true at the latest during the $\sigma$th loop iteration from time $t'$ and $s$ sends ALIVE to $p$ in the same loop iteration. Hence, from time $t'$, $s$ sends ALIVE to $p$ in at most $\sigma \times \beta$ times, i.e., in at most $\delta$ times. As $t' \leq t + \beta$, the lemma is proven. $\Box$

As all the output links of $s$ are timely, we have the next observation:

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 loop iteration after the delivery of $m$. Hence, by Observations 5 (page 7) and 11, follows:

Lemma 35  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 $[t, t + \delta + 2\beta]$.

By Lemmas 34 and 35, follows:

Lemma 36  For every alive process $p \neq s$, $p$ receives ALIVE from $s$ at least every $2\delta + 3\beta$ times.

Lemma 37  For every alive process $p$, there is a time after which $s \in \text{Actives}_p$ forever.

Proof.  First, the lemma trivially holds for $p = s$. Consider now the case where $p \neq s$. There is a time after which $s \in \text{Actives}_p$ forever if and only if there is a time after which $s \in \text{OtherActives}_p$ forever. We know that $\text{OtherActives}_p$ is periodically reset to $\text{Collect}_p$ and, after that, $\text{Collect}_p$ is reset to $\emptyset$. After such a reset, $p$ waits $5\delta$ complete loop iterations, i.e. at least $5\delta$ times, before setting $\text{OtherActives}_p$ to $\text{Collect}_p$ again. During this period, $p$ receives at least one ALIVE message from $s$ by Lemma 36. So, during this period, $p$ inserts $s$ in $\text{Collect}_p$. Hence, when $p$ sets $\text{OtherActives}_p$ to $\text{Collect}_p$ again, $s \in \text{Collect}_p$. $\Box$

Lemma 38  In any execution, $\text{Counter}_s[s]$ is bounded.

Proof.  Assume, by contradiction, that $\text{Counter}_s[s]$ increases infinitely often. So, $s$ receives ACCUSE messages infinitely often: there is at least one alive process $p \neq s$ that accuses $s$ infinitely often. Now, $p$ only sends ACCUSE messages to processes $q$ such that $q \notin \text{Actives}_p$ and $s \in \text{Actives}_p$ eventually holds
forever by Lemma 37 — a contradiction.

Definition 3 For each process $p$, let $c_p$ be the largest value of $\text{Counter}_p[p]$ in the execution that we consider ($c_p = \infty$ if $\text{Counter}_p[p]$ is unbounded). Let $\ell$ be the process such that $(c_\ell, \ell) = \min\{(c_p, p) : p \text{ is an alive process}\}$.

By Definition, $\ell$ is an alive process. Furthermore, by Lemma 38, $c_\ell < \infty$, so, $c_\ell < \infty$, i.e., $\text{Counter}_\ell[\ell]$ is bounded.

Lemma 39 Let $p$ and $q$ be two alive processes. The two following propositions holds:

(a) if $q \in \text{Actives}_p$ infinitely often and $c_q < \infty$, then there is a time after which $\text{Counter}_p[q] = c_q$ forever.
(b) if $q \in \text{Actives}_p$ infinitely often and $c_q = \infty$, then there is a time after which $\text{Counter}_p[q] > c_\ell$ forever.

Proof. First, if $p = q$, then (a) holds because $\text{Counter}_q[q]$ is eventually monotonically nondecreasing by Observation 10. Then, if $p = q$, then (b) holds because $\text{Counter}_q[q]$ is eventually monotonically nondecreasing and $c_\ell$ is bounded.

Consider now the case where $p \neq q$. In the two cases (a) and (b), $p$ receives $\text{ALIVE}$ from $q$ infinitely often by Lemma 31.

(a) Assume now that $c_q < \infty$. In this case, $\text{Counter}_q[q]$ is bounded and eventually monotonically nondecreasing (Observation 10). So, there is a time $t$ after which $\text{Counter}_q[q] = c_q$ 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 $q$ $\text{ALIVE}$ messages that have been sent by $q$ after time $t$ and all these messages are of the following form: $(\text{ALIVE}, -,-, c_q)$. Now, each time $p$ receives such an $(\text{ALIVE}, -,-, c_q)$ message, $p$ sets $\text{Counter}_p[q]$ to $c_q$. Hence, there is a time after which $\text{Counter}_p[q] = c_q$ forever.

(b) Assume that $c_q = \infty$. In this case, $\text{Counter}_q[q]$ is unbounded. Then, we already know that $\text{Counter}_\ell[\ell]$ is bounded. So, there is a time after which $\text{Counter}_q[q] > \text{Counter}_\ell[\ell]$ forever (remember that $\text{Counter}_q[q]$ and $\text{Counter}_\ell[\ell]$ are eventually monotonically nondecreasing by Observation 10). Thus, by Lemma 33, there is a time after which $\text{Counter}_p[q] \geq \text{Counter}_q[q] > \text{Counter}_\ell[\ell]$ forever. Now, $\text{Counter}_\ell[\ell]$ is eventually equal to $c_\ell$ forever because $\text{Counter}_\ell[\ell]$ is monotonically nondecreasing. Hence, there is a time after which $\text{Counter}_p[q] > c_\ell$ forever.

As $\text{LLeader}_p[p]$ is regularly set to $q$ such that $q \in \text{Actives}_p$, we have:

Corollary 4 Let $p$ and $q$ be two alive processes, the two following propositions holds:
(a) if $\text{LLeader}_p[q] = q$ infinitely often and $c_q < \infty$, then there is a time after which $\text{Counter}_p[q] = c_q$ forever.

(b) if $\text{LLeader}_p[q] = q$ infinitely often and $c_q = \infty$, then there is a time after which $\text{Counter}_p[q] > c_q$ forever.

**Lemma 40** Let $p$ be an alive process. Let $q$ be a process. Assume that $q \in \text{Actives}_p$ and $\text{LLeader}_p[q] = r$ holds infinitely often. The two following propositions hold:

(a) There is a time after which the predicate ($\text{LLeader}_p[q] = r$) implies ($\text{LLCounter}_p[q] = c_r$) holds each time $p$ sets $\text{Leader}_p$ to $\text{MyLeader}_p$, if $c_r < \infty$.

(b) There is a time after which the predicate ($\text{LLeader}_p[q] = r$) implies ($\text{LLCounter}_p[q] > c_r$) holds each time $p$ sets $\text{Leader}_p$ to $\text{MyLeader}_p$, if $c_r = \infty$.

**Proof.** Assume that $p = q$. By Corollary 4, there is a time after which:

- $\text{Counter}_p[r] = c_r$ forever, if $c_r < \infty$
- $\text{Counter}_p[r] > c_r$ forever, if $c_r = \infty$

So, the lemma holds because $p$ periodically updates $\text{LLeader}_p[p]$, sets $\text{LLCounter}_p[p]$ to $\text{Counter}_p[\text{LLeader}_p[p]]$, and sets $\text{Leader}_p$ to $\text{MyLeader}_p$.

Assume now that $q \neq p$. Then, by Lemmas 31 and 32, $p$ receives $\text{ALIVE}$ messages from $q$ infinitely often and $q$ is alive. As the number of messages initially in the link $(q,p)$ is finite, eventually $p$ only receives from $q$ $\text{ALIVE}$ messages sent by $q$. Each $\text{ALIVE}$ message sent by $q$ at time $t$ is of the following form: $(\text{ALIVE},v,\text{vcnt},\text{gcnt})$ where $v$ is the value of $\text{LLeader}_q[q]$ at time $t$ and $\text{vcnt}$ is the value of $\text{Counter}_q[\text{LLeader}_q[q]]$ at time $t$. When receiving such a message, $p$ sets $\text{LLeader}_p[q]$ to $v$ and $\text{LLCounter}_p[q]$ to $\text{vcnt}$ in sequel. Moreover, this is the only way to modify $\text{LLeader}_p[q]$ and $\text{LLCounter}_p[q]$. Thus, $\text{LLeader}_p[q] = r$ holds infinitely often implies that $\text{LLeader}_q[q] = r$ holds infinitely often and, by Corollary 4:

- if $c_r < \infty$, then $\text{Counter}_q[r] = c_r$ eventually holds forever.
- if $c_r = \infty$, then $\text{Counter}_q[r] > c_r$ eventually holds forever.

If $c_r < \infty$, then eventually $p$ only receives from $q$ ($\text{ALIVE},v,\text{vcnt},\text{gcnt}$) messages that satisfy the condition ($v = r$) \Rightarrow ($\text{vcnt} = c_r$). At each reception of such messages, $p$ sets $\text{LLeader}_p[q]$ to $r$ and $\text{LLCounter}_p[q]$ to $c_r$ in sequel. So, eventually each time $p$ sets $\text{Leader}_p$ to $\text{MyLeader}_p$, we have $\text{LLCounter}_p[q] = c_r$, if $\text{LLeader}_p[q] = r$ and Part (a) of the lemma is proven.

If $c_r = \infty$, then eventually $p$ only receives from $q$ ($\text{ALIVE},v,\text{vcnt},\text{gcnt}$) messages that satisfy the condition ($v = r$) \Rightarrow ($\text{vcnt} > c_r$). At each reception of such messages, $p$ sets $\text{LLeader}_p[q]$ to $r$ and $\text{LLCounter}_p[q]$ to $c_r$ in sequel. So, eventually each time $p$ sets $\text{Leader}_p$ to $\text{MyLeader}_p$, we have $\text{LLCounter}_p[q] > c_r$, if $\text{LLeader}_p[q] = r$ and Part (b) of the lemma is proven. 

The following lemma is straightforward by definition of $\ell$ and by the way $p$
computes $LLeader_p[p]$.

**Lemma 41** For every alive process $p$, if there is a time after which $\ell \in Actives_p$ forever, then there is a time after which $LLeader_p[p] = \ell$ forever.

**Definition 4** Let $LLLeaders(p) = \{LLeader_p[q] : q \in Actives_p\}$.

**Lemma 42** For every alive process $p$, if there is a time after which $\ell \in LLLeaders(p)$ forever, then there is a time after which $Leader_p = \ell$ forever.

**Proof.** Assume that there is a time after which $\ell \in LLLeaders(p)$ forever. Then, as $\ell \in LLLeaders(p)$ holds infinitely often and $LLLeaders(p) = \{LLeader_p[q] : q \in Actives_p\}$, there is a subset of processes $S$ such that:

1. $\forall q \in S, q \in Actives_p$ and $LLeader_p[q] = \ell$ holds infinitely often.

Also, as there is a time $t$ after which $\ell \in LLLeaders(p)$ forever, we have the following additional property:

2. $\forall t' \geq t, \exists q_{t'} \in S$ such that $q_{t'} \in Actives_p$ and $LLeader_p[q_{t'}] = \ell$ at time $t'$.

By (1) and Lemma 40, there is a time after which $\forall q \in S, (LLeader_p[q] = \ell) \Rightarrow (LLC\text{ounter}_p[q] = c_t))$ 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 time $t' \geq t$, then there exists a process $q_{t'} \in S$ such that $LLeader_p[q_{t'}] = \ell$ and $LLCounter_p[q_{t'}] = c_t$ at time $t'$.

Assume now, by contradiction, that $Leader_p \neq \ell$ infinitely often. Then, as $Leader_p$ is periodically set to $MyLeader_p$, the following situation appears infinitely often: $p$ sets $Leader_p$ to $MyLeader_p$ while there exists two processes $v$ and $r$ such that $v \in Actives_p$, $LLeader_p[v] = r$, and $(LLCounter_p[v], LLeader_p[v]) < (c_t, \ell)$. Two cases are then possible:

- $c_r < \infty$. Then, there is a time after which the condition $(LLeader_p[v] = r) \Rightarrow (LLCounter_p[v] = c_r)$ holds each time $p$ sets $Leader_p$ to $MyLeader_p$, by Part (a) of Lemma 40. Now, by Definition $(c_r, r) > (c_t, \ell)$. So, $(LLCounter_p[v], LLeader_p[v]) > (c_t, \ell)$ eventually holds each time $p$ sets $Leader_p$ to $MyLeader_p$ while $v \in Actives_p$ and $LLeader_p[v] = r$ — a contradiction.

- $c_r = \infty$. Then, there is a time after which the condition $(LLeader_p[v] = r) \Rightarrow (LLCounter_p[v] > c_t)$ holds each time $p$ sets $Leader_p$ to $MyLeader_p$, by Part (b) of Lemma 40. So, $(LLCounter_p[v], LLeader_p[v]) > (c_t, \ell)$ eventually holds each time $p$ sets $Leader_p$ to $MyLeader_p$ while $v \in Actives_p$ and $LLeader_p[v] = r$ — a contradiction.

We now show that for every alive process $p$ there is a time after which $\ell \in LLLeaders(p)$.
Lemma 43 There is a time after which $\ell \in \text{Actives}_s$ forever.

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

In Case (2), $\ell$ is removed from $\text{Actives}_s$ each time $\ell$ was in $\text{Actives}_s$ but not in $\text{Collect}_s$ and $s$ sets $\text{OtherActives}_s$ to $\text{Collect}_s$. In this case, $s$ sends an ACCUSE message to $\ell$. So, $s$ sends ACCUSE messages to $\ell$ infinitely often.

In Case (3), as there is a time after which $\ell \notin \text{Actives}_s$ forever and as $s$ periodically sends ACCUSE messages to every process $q$ such that $q \notin \text{Actives}_s$, $s$ sends ACCUSE messages to $\ell$ infinitely often.

So, in both Cases (2) and (3), $s$ sends ACCUSE messages to $\ell$ infinitely often. Now, since the output links of $s$ are timely and $\ell$ tries to receive ACCUSE messages from $s$ infinitely often, $\ell$ receives ACCUSE messages from $s$ infinitely often. Thus, $\ell$ increments $\text{Counter}_\ell[\ell]$ infinitely often and, as $\text{Counter}_\ell[\ell]$ is eventually monotonically nondecreasing (Observation 10), $\text{Counter}_\ell[\ell]$ unbounded — a contradiction. Hence, only Case (1) is possible.

By Lemmas 41 and 43, follows:

Lemma 44 There is a time from which $\text{LLeader}_s[s] = \ell$ forever.

Lemma 45 For every alive process $p$, $\text{LLeader}_p[s] = \ell$ eventually holds forever.

Proof. Let $p$ be an alive process. If $p = s$, then the result is immediate from Lemma 44. Assume now that $p \neq s$. In this case, $p$ receives ALIVE messages from $s$ infinitely often by Lemma 36. By Lemma 44, there is a time $t$ after which $\text{LLeader}_s[s] = \ell$. So, after time $t$, all the ALIVE messages that $s$ sends to $p$ are of the form (ALIVE,$\ell$,−,−). Thus, there is a time after which all the ALIVE messages that $p$ receives from $s$ are of the form (ALIVE,$\ell$,−,−). So, there is a time after which $\text{LLeader}_p[s] = \ell$ forever.

By Lemmas 37, 45, and Definition 4, follows:

Corollary 5 Each alive process $p$ eventually satisfies $\ell \in \text{LLeaders}(p)$ forever.

By Corollary 5 and Lemma 42, follows:

Lemma 46 For every alive process $p$, there is a time after which $\text{Leader}_p = \ell$
forever.

By Lemma 46 and the fact that \( \ell \) is alive, follows:

**Theorem 9** Algorithm 4 implements a PSSCF leader election in system \( S_1 \).

### 7 Conclusion and Future Works

We considered stabilizing leader election in various partial synchronous systems where any process can crash. Figure 2 summarizes our results. This work exhibits some assumptions that are required to obtain CE-SSCF, SSSCF, CE-PSSCF, and PSSCF leader election algorithms. In particular, it emphasizes that PSSCF is easier to achieve than SSSCF: PSSCF solutions require weaker assumptions than SSSCF solutions. Finally, this article show that for silent tasks [15] such as leader election the gap between fault-tolerance and fault-tolerant pseudo-stabilization is not really significant: in such problems, adding a pseudo-stabilizing property to a fault-tolerant algorithm is quite easy.

There are some possible extensions to this work. First, we can study stabilizing leader election in systems where only a given number of processes may crash. Then, it could be interesting to extend these algorithms and results to other communication topologies. Finally, we can study the implementability of stabilizing solutions in systems with crash failures for other classes of problems such as decision problems.

<table>
<thead>
<tr>
<th></th>
<th>( S_4 )</th>
<th>( S_3 )</th>
<th>( S_2 )</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication-Efficient Self-Stabilization (CE-SSCF)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Self-Stabilization (SSCF)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Communication-Efficient Pseudo-Stabilization (CE-PSSCF)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Pseudo-Stabilization (PSSCF)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Fig. 2. Implementability of stabilizing leader election.

### References


