Self-Stabilizing Leader Election in Polynomial Steps*

Karine Altisen[1], Alain Cournier[2], Stéphane Devismes[1], Anaïs Durand[1], and Franck Petit[3]

[1] VERIMAG UMR 5104, Université Grenoble Alpes, France
[2] MIS Lab., Université Picardie Jules Verne, France
[3] LIP6 UMR 7606, INRIA, UPMC Sorbonne Universités, France

Abstract. In this paper, we propose a silent self-stabilizing leader election algorithm for bidirectional connected identified networks of arbitrary topology. This algorithm is written in the locally shared memory model. It assumes the distributed unfair daemon, the most general scheduling hypothesis of the model. Our algorithm requires no global knowledge on the network (such as an upper bound on the diameter or the number of processes, for example). We show that its stabilization time is in $\Theta(n^3)$ steps in the worst case, where n is the number of processes. Its memory requirement is asymptotically optimal, *i.e.*, $\Theta(\log n)$ bits per processes. Its round complexity is of the same order of magnitude — *i.e.*, $\Theta(n)$ rounds — as the best existing algorithms [10, 9] designed with similar settings. To the best of our knowledge, this is the first asynchronous self-stabilizing leader election algorithm for arbitrary identified networks that is proven to achieve a stabilization time polynomial in steps. By contrast, we show that the previous best existing algorithms designed with similar settings [10, **?**] stabilize in a non polynomial number of steps in the worst case.

1 Introduction

In distributed computing, the *leader election* problem consists in distinguishing a single process, so-called the leader, among the others. We consider here identified networks. So, as it is usually done, we augment the problem by requiring all processes to eventually know the identifier of the leader. The leader election is fundamental as it is a basic component to solve many other important problems, *e.g.*, consensus, spanning tree constructions, implementing broadcasting and convergecasting methods, *etc*.

Self-stabilization [11] is a versatile technique to withstand *any* transient fault in a distributed system: a self-stabilizing algorithm is able to recover, *i.e.*, reach a legitimate configuration, in finite time, regardless the *arbitrary* initial configuration of the system, and therefore also after the occurrence of transient faults. Thus, self-stabilization makes no hypotheses on the nature or extent of transient faults that could hit the system, and recovers from the effects of those faults in a unified manner. Such versatility comes at a price. After transient faults, there is a finite period of time, called the *stabilization phase*, before the

* This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir and the AGIR project DIAMS.

system returns to a legitimate configuration. The *stabilization time* is then the maximum time to reach a legitimate configuration starting from an arbitrary one. Notice that efficiency of self-stabilizing algorithms is mainly evaluated according to their stabilization time and memory requirement.

We consider deterministicasynchronous silent self-stabilizing leader election problem in bidirectional, connected, and identified networks of arbitrary topology. We investigate solutions to this problem which are written in the locally shared memory model introduced by Dijkstra [11]. In this model, the distributed unfair daemon is known as the weakest scheduling assumption. Under such an assumption, proving that a given algorithm is self-stabilizing implies that the stabilization time must be finite in terms of atomic steps. However, despite some solutions assuming all these settings (in particular the unfairness assumption) are available in the literature [10, **?**,**?**], none of them is proven to achieve a polynomial upper bound in steps on its stabilization time. Actually, the time complexities of all these solutions are analyzed in terms of rounds only.

Related Work In [12], Dolev *et al* showed that silent self-stabilizing leader election requires $\Omega(\log n)$ bits per process, where n is the number of processes. Notice that *non-silent* self-stabilizing leader election can be achieved using less memory, *e.g.*, the non-silent self-stabilizing leader election algorithm for unoriented ring-shaped networks given in [5] requires $O(\log \log n)$ space per process.

Self-stabilizing leader election algorithms for arbitrary connected identified networks have been proposed in the message-passing model [1, 4, 6]. First, the algorithm of Afek and Bremler [1] stabilizes in $O(n)$ rounds using $\Theta(\log n)$ bits per process. But, it assumes that the link-capacity is bounded by a value B, known by all processes. Two solutions that stabilize in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network, have been proposed in [4, 6]. However, both solutions assume that processes know some upper bound D on the diameter $\mathcal{D}$; and require $\Theta(\log D \log n)$ bits per process.

Several solutions are also given in the shared memory model [13, 3, 8, 10, 9, 14]. The algorithm proposed by Dolev and Herman [13] is not silent, works under a *fair* daemon, and assume that all processes know a bound N on the number of processes. This solution stabilizes in $O(\mathcal{D})$ rounds using $\Theta(N \log N)$ bits per process. The algorithm of Arora and Gouda [3] works under a *weakly fair* daemon and assume the knowledge of some bound N on the number of processes. This solution stabilizes in $O(N)$ rounds using $\Theta(\log N)$ bits per process.

Datta *et al* [8] propose the first self-stabilizing leader election algorithm (for arbitrary connected identified networks) proven under the distributed unfair daemon. This algorithm stabilizes in $O(\mathcal{D})$ rounds. However, the space complexity of this algorithm is unbounded. (More precisely, the algorithm requires each process to maintain an unbounded integer in its local memory.)

Solutions in [10, 9, 14] have a memory requirement which is asymptotically optimal (*i.e.* in $\Theta(\log n)$). The algorithm proposed by Kravchik and Kutten [14] assumes a synchronous daemon and the stabilization time of this latter is in $O(\mathcal{D})$ rounds. The two solutions proposed by Datta *et al* in [10, 9] assume a distributed unfair daemon and have a stabilization time in $O(n)$ rounds. However, despite these two algorithms stabilizing within a finite number of steps (indeed, they are proven assuming an unfair daemon), no step complexity analysis is proposed.

Contribution We propose a silent self-stabilizing leader election algorithm for arbitrary connected and identified networks. Our solution is written in the locally shared memory model assuming a distributed unfair daemon, the weakest scheduling assumption. Our algorithm assumes no knowledge of any global parameter (*e.g.*, an upper bound on $\mathcal{D}$ or n) of the network. Like previous solutions of the literature [10, 9], it is asymptotically optimal in space (*i.e.*, it can be implemented using $\Theta(\log n)$ bits per process), and it stabilizes in $\Theta(n)$ rounds in the worst case. Yet, contrary to those solutions, we show that our algorithm has a stabilization time in $\Theta(n^3)$ steps in the worst case.

For fair comparison, we have also studied the step complexity of the algorithms given in [10, **?**], noted here $\mathcal{DLV}$ and $\mathcal{DLV}2$, respectively. These latter are the closest to ours in terms of performance. We show that their stabilization time is not polynomial. Indeed, there is no constant α such that the stabilization time of $\mathcal{DLV}$ is in $O(n^\alpha)$ steps. More precisely, we show that fixing α to any constant greater than or equal to 4, for every $\beta \geq 2$, there exists a network of $n = 2^{\alpha-1} \times \beta$ processes in which there exists a possible execution that stabilizes in $\Omega(n^\alpha)$ steps. Similarly, for $n \geq 5$, there exists a network and a possible execution of $\mathcal{DLV}2$ that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps. Due to the lack of space, this latter result is not presented here. Refer to the technical report online [2] for more details.

Roadmap. The next section is dedicated to computational model and basic definitions. In Section 3, we propose our self-stabilizing leader election algorithm. In Section 4, we outline the proof of correctness and the complexity analysis. A detailed proof of correctness and a complete complexity analysis are available in the technical report online [2]. Finally, we conclude in Section 5.

2 Computational Model

Distributed systems. We consider *distributed systems* made of n *processes*. Each process can communicate with a subset of other processes, called its *neighbors*. We denote by $\mathcal{N}_p$ the set of neighbors of process p. Communications are assumed to be bidirectional, *i.e.* $q \in \mathcal{N}_p$ if and only if $p \in \mathcal{N}_q$. Hence, the topology of the system can be represented as a simple undirected connected graph

$G = (V, E)$, where V is the set of processes and E is a set of edges representing (direct) communication relations. We assume that each process has a unique ID, a natural integer. IDs are stored using a constant number of bits, b. As commonly done in the literature, we assume that $b = \Theta(\log n)$. Moreover, by an abuse of notation, we identify a process with its ID, whenever convenient. We will also denote by ℓ the process of minimum ID. (So, the minimum ID will be also denoted by ℓ.)

Locally shared memory model. We consider the *locally shared memory model* in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can only write to its own variables. The *state* of a process is the vector of values of all its variables. A configuration γ of the system is a vector consisting in one state of each process. We denote by $\mathcal{C}$ the set of all possible configurations.

A distributed *algorithm* consists of one *program* per process. The program of a process p is a finite set of *actions* of the following form: $\langle$label$\rangle$ :: $\langle$guard$\rangle \rightarrow \langle$statement$\rangle$. The *labels* are used to identify actions. The *guard* of an action in the program of process p is a Boolean expression involving the variables of p and its neighbors. If the guard of some action evaluates to true, then the action is said to be *enabled* at p. By extension, if at least one action is enabled at p, p is said to be enabled. We denote by $Enabled(\gamma)$ the set of processes enabled in configuration γ. The *statement* of an action is a sequence of assignments on the variables of p. An action can be executed only if it is enabled. In this case, the execution of the action consists in executing its statement.

The asynchronism of the system is materialized by an adversary, called the *daemon*. In a configuration γ, if there is at least one enabled process, then the daemon selects a non empty subset S of $Enabled(\gamma)$ to perform an *(atomic) step*: $\forall p \in S$, p atomically executes one of its actions enabled in γ, leading the system to a new configuration γ'. We denote by $\mapsto$ the relation between configurations such that $\gamma \mapsto \gamma'$ if and only if γ' can be reached from γ in one (atomic) step. An *execution* is then a *maximal* sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i, \forall i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration γ in which $Enabled(\gamma)$ is empty.

In this paper, the daemon is supposed to be *distributed* and *unfair*. "Distributed" means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. "Unfair" means that there is no fairness constraint, *i.e.*, the daemon might never permit an enabled process to execute, unless it is the only enabled process.

Rounds. To measure the time complexity of an algorithm, we also use the notion of *round* [**?**]. This latter allows to highlight the execution time according to the speed of the slowest process. If a process p is enabled in a configuration γ_i but not enabled in the next configuration γ_{i+1} and does not execute any action between γ_i and γ_{i+1}, we said that p is *neutralized* during the step $\gamma_i \mapsto \gamma_{i+1}$. The first round of an execution e, noted e', is the minimal prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let e'' be the suffix of e starting from the last configuration of e'. The second round of e is the first round of e'', and so forth.

Self-stabilization. Let $\mathcal{A}$ be a distributed algorithm. Let $\mathcal{E}$ be the set of all possible executions of $\mathcal{A}$. A *specification* SP is a predicate over $\mathcal{E}$.

$\mathcal{A}$ is *self-stabilizing* for SP if and only if there exists a non-empty subset of configurations $\mathcal{L} \subseteq \mathcal{C}$, called *legitimate* configurations, such that:

- *Closure:* $\forall e \in \mathcal{E}$, for each step $\gamma_i \mapsto \gamma_{i+1} \in e$, $\gamma_i \in \mathcal{L} \Rightarrow \gamma_{i+1} \in \mathcal{L}$.
- *Convergence:* $\forall e \in \mathcal{E}, \exists \gamma \in e$ such that $\gamma \in \mathcal{L}$.
- *Correctness:* $\forall e \in \mathcal{E}$ such that e starts in a legitimate configuration $\gamma \in \mathcal{L}$, e satisfies SP.

Every configuration that is not legitimate is called *illegitimate*. The *stabilization time* is the maximum time (in steps or rounds) to reach a legitimate configuration starting from any configuration.

Self-stabilizing leader election. We define SP_{LE} the specification of the leader election problem. Let $Leader : V \mapsto \mathbb{N}$ be a function defined on the state of any process $p \in V$ in the current configuration that returns the ID of the leader appointed by p. An execution $e \in \mathcal{E}$ satisfies SP_{LE} if and only if:

1. For each configuration $\gamma \in e$, $\forall p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process in V.
2. For each step $\gamma_i \mapsto \gamma_{i+1} \in e$, $\forall p \in V$, $Leader(p)$ has the same value in γ_i and γ_{i+1}.

An algorithm $\mathcal{A}$ is *silent* if and only if every execution is finite [12]. Let γ be a terminal configuration. The set of all possible executions starting from γ is the singleton $\{\gamma\}$. So, if $\mathcal{A}$ is self-stabilizing and silent, γ must be legitimate. Thus, to prove that a leader election algorithm is both self-stabilizing and silent, it is necessary and sufficient to show that: (1) in every terminal configuration γ, $\forall p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process; (2) every execution is finite.

3 Algorithm $\mathcal{LE}$

In this section, we present a silent and self-stabilizing leader election algorithm, called $\mathcal{LE}$. Its formal code is given in Algorithm 1. Starting from an arbitrary

configuration, $\mathcal{LE}$ converges to a terminal configuration, where the process of minimum ID, ℓ, is elected. More precisely, in the terminal configuration, every process p knows the identifier of ℓ thanks to its local variable $p.idR$. This means that, in particular, we instantiate the function $Leader$ of the specification as follows: $Leader(p) = p.idR$, $\forall p \in V$. Moreover, a spanning tree rooted at ℓ is defined using two variables per process: par and $level$. First, $\ell.par = \ell$ and $\ell.level = 0$. Then, for every process $p \neq \ell$, $p.par$ points to the parent of p in the tree and $p.level$ is the level of p in the tree.

We now present a simple algorithm for the leader election in Subsection 3.1. We show why this algorithm is not self-stabilizing in Subsection 3.2. We explain in Subsection 3.3 how to modify this algorithm to make it self-stabilizing.

3.1 Non Self-Stabilizing Leader Election

We first consider a simplified version of $\mathcal{LE}$. Starting from a predefined initial configuration, it elects ℓ in all idR variables and builds a spanning tree rooted at ℓ. Initially, every process p declares itself as leader: $p.idR = p$, $p.par = p$, and $p.level = 0$. So, p satisfies the two following predicates: $SelfRoot(p) \equiv (p.par = p)$ and $SelfRootOk'(p) \equiv (p.level = 0) \wedge (p.idR = p)$. Note that, in the sequel, we say that p is a *self root* when $SelfRoot(p)$ holds. From such an initial configuration, our non self-stabilizing algorithm consists in the following single action:

$$\begin{aligned} J\text{-}Action' :: \exists q \in \mathcal{N}_p, (q.idR < p.idR) \rightarrow\ & p.par \leftarrow \min_{\preceq}\{q \in \mathcal{N}_p\}; \\ & p.idR \leftarrow p.par.idR; \\ & p.level \leftarrow p.par.level + 1; \end{aligned}$$

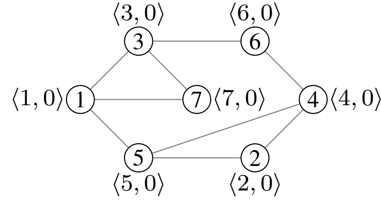
where $\forall x, y \in V, x \preceq y \Leftrightarrow (x.idR \leq y.idR) \wedge [(x.idR = y.idR) \Rightarrow (x < y)]$

Informally, when p discovers that $p.idR$ is not equal to the minimum identifier, it updates its variables accordingly. Let q be the neighbor of p having idR minimum. Then, p selects q as new parent ($p.par \leftarrow q$ and $p.level \leftarrow p.par.level + 1$) and sets $p.idR$ to the value of $q.idR$. If there are several neighbors having idR minimum, the identifiers of those neighbors are used to break ties.

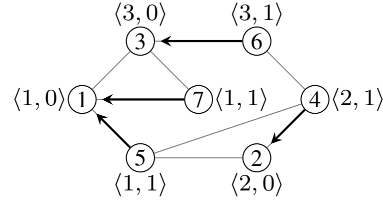
Hence, the identifier of ℓ is propagated, from neighbors to neighbors, into the idR variables and the system reaches a terminal configuration in $O(\mathcal{D})$ rounds. Figure 1 shows an example of such an execution.

Notice first that for every process p, $p.idR$ is always less than or equal to its own identifier. Indeed, $p.idR$ is initialized to p and decreases each time p executes $J\text{-}Action'$. Hence, $p.idR = p$ while p is a self root and after p executes $J\text{-}Action'$ for the first time, $p.idR$ is smaller than its ID forever.

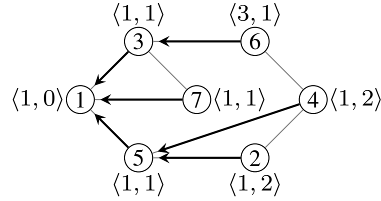
Second, even in this simplified context, for each two neighbors p and q such that q is the parent of p, it may happen that $p.idR$ is greater than $q.idR$ —an example is shown in Figure 1c, where $p = 6$ and $q = 3$. This is due to the fact



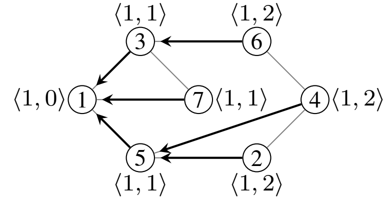
(a) Initial configuration. $SelfRoot(p) \wedge SelfRootOk'(p)$ holds for every process p.



(b) 4, 5, 6, and 7 have executed $J\text{-}Action'$. Note that $J\text{-}Action'$ was not enabled at 2 because it is a local minimum.



(c) 2, 3, and 4 have executed $J\text{-}Action'$. 3 joins the tree rooted at 1, but the new value of $3.idR$ is not yet propagated to its child 6.



(d) 6 has executed $J\text{-}Action'$. The configuration is now terminal, $\ell = 1$ is elected, and a tree rooted at ℓ is available.

Fig. 1: An example showing an execution of the non self-stabilizing algorithm. Process identifiers are given inside the nodes. $\langle x, y \rangle$ means $idR = x$ and $level = y$. Arrows represent par pointers. The absence of arrow means that the process is a self root.

that p joins the tree of q but meanwhile q joins another tree and this change is not yet propagated to p. Similarly, when $p.idR \neq q.idR$, $p.level$ may be different from $q.level + 1$.

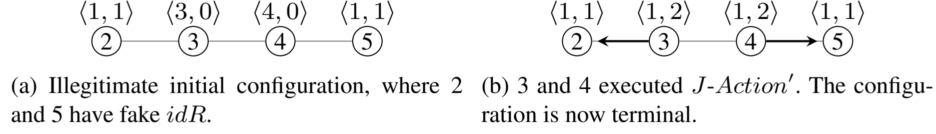
According to those remarks, we can deduce that when $p.par = q$ with $q \neq p$, we have the following relation between p and q:

$$GoodIdR(p, q) \equiv (p.idR \geq q.idR) \wedge (p.idR < p)$$
$$GoodLevel(p, q) \equiv (p.idR = q.idR) \Rightarrow (p.level = q.level + 1)$$

3.2 Fake IDs

The algorithm presented in Subsection 3.1 is clearly not self-stabilizing. Indeed, in a self-stabilization context, the execution may start in any arbitrary configuration. In particular, idR variables can be initialized to arbitrary natural integer values, even values that are actually not IDs of (existing) processes. We call such values *fake IDs*.

The existence of fake IDs may lead the system to an illegitimate terminal configuration. Refer to the example of execution given in Figure 2: starting from the configuration in 2a, if processes 3 and 4 move, the system reaches the terminal configuration given in 2b, where there are two trees and the idR variables



(a) Illegitimate initial configuration, where 2 and 5 have fake idR.

(b) 3 and 4 executed $J\text{-}Action'$. The configuration is now terminal.

Fig. 2: Example of execution that does not converge to a legitimate configuration.

elect the fake ID 1. In this example, 2 and 5 can detect the problem. Indeed, predicate $SelfRootOk'$ is violated by both 2 and 5. One may believe that it is sufficient to reset the local state of processes which detect inconsistency (here processes 2 and 5) to $p.idR \leftarrow p$, $p.par \leftarrow p$ and $p.level \leftarrow 0$. After these resets, there are still some errors, as shown on Figure 3. Again, 3 and 4 can detect the problem. Indeed, predicate $GoodIdR(p, p.par) \wedge GoodLevel(p, p.par)$ is violated by both 3 and 4. In this example, after 3 and 4 have reset, all inconsistencies have been removed. So let define the following action:

$$\begin{array}{lll} R\text{-}Action' :: & \big(SelfRoot(p) \wedge \neg SelfRootOk'(p)\big) \vee \big(\neg SelfRoot(p) & \rightarrow p.par \leftarrow p; \\ & \wedge \neg(GoodIdR(p, p.par) \wedge GoodLevel(p, p.par))\big) & \quad p.idR \leftarrow p; \\ & & \quad p.level \leftarrow 0; \end{array}$$

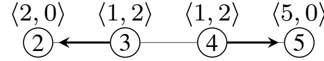
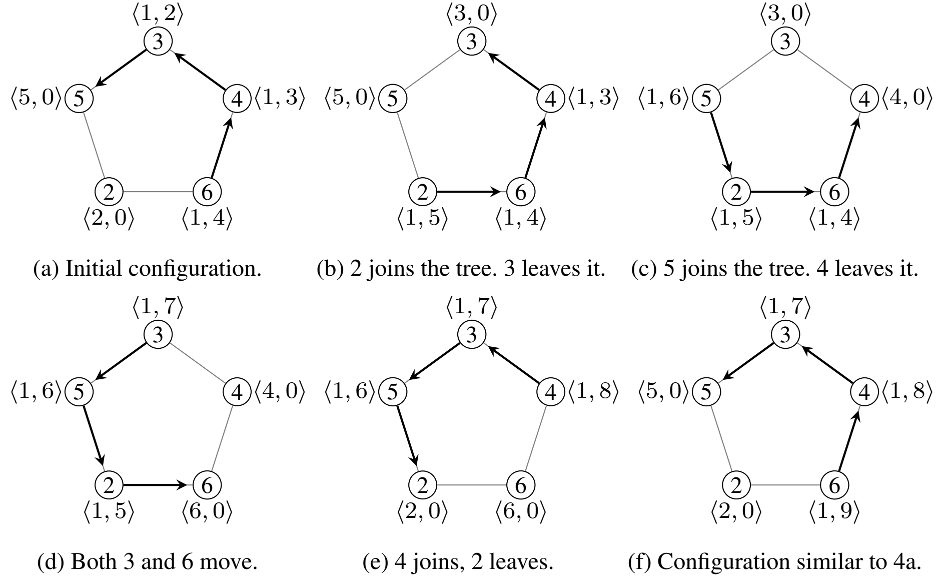


Fig. 3: One step after Figure 2b, 2 and 5 have reset.

Unfortunately, this additional action does not ensure the convergence in all cases—refer to the example in Figure 4. Indeed, if a process resets, it becomes a self root but this does not erase the fake ID in the rest of its subtree. Then, another process can join the tree and adopt the fake ID which will be further propagated, and so on. In the example, a process resets while another joins its tree at lower level, and this leads to endless erroneous behavior, since we do not want to assume any maximal value for $level$ (such an assumption would otherwise imply the knowledge of some upper bound on n). Therefore, the whole tree must be reset, instead of its root only. To that goal, we first freeze the "abnormal" tree in order to forbid any process to join it, then the tree is reset top-down. The cleaning mechanism is detailed in the next subsection.

3.3 Cleaning Abnormal Trees

To introduce the trees, we define what is a "good relation" between a parent and its children. Namely, the predicate $KinshipOk'(p, q)$ models that a process p is a *real child* of its parent $q = p.par$. This predicate holds if and only if $GoodLevel(p, q)$ and $GoodIdR(p, q)$ are true. This relation defines a spanning forest: a *tree* is a maximal set of processes connected by par pointers and satisfying $KinshipOk'$ relation. A process p is a root of such a tree whenever



(a) Initial configuration. (b) 2 joins the tree. 3 leaves it. (c) 5 joins the tree. 4 leaves it.

(d) Both 3 and 6 move. (e) 4 joins, 2 leaves. (f) Configuration similar to 4a.

Fig. 4: The first process of the chain of bold arrows violates the predicate $SelfRootOk'$ and resets by executing $R\text{-}Action'$, while another process joins its tree. This cycle of resets and joins might never terminate.

$SelfRoot(p)$ holds or $KinshipOk'(p, p.par)$ is false. When $SelfRoot(p) \wedge SelfRootOk'(p)$ is true, p is a *normal root* just as in the non self-stabilizing case. In other cases, there is an error and p is called an *abnormal root*: $AbRoot'(p) \equiv \big(SelfRoot(p) \wedge \neg SelfRootOk'(p)\big) \vee \big(\neg SelfRoot(p) \wedge \neg KinshipOk'(p, p.par)\big)$. A tree is said to be *abnormal* (resp. *normal*) when its root is abnormal (resp. normal).

We now detail the different predicates and actions of Algorithm 1.

Variable $status$. Abnormal trees need to be frozen before to be cleaned in order to prevent them from growing endlessly (see 3.2). This mechanism is achieved using an additional variable, $status$, that is used as follows. If a process is clean (*i.e.*, not involved into any freezing operation), then its $status$ is C. Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter states are actually used to perform a "Propagation of Information with Feedback" [7] into the abnormal trees. Status EB means "Error Broadcast" and EF means "Error Feedback". From an abnormal root, the status EB is broadcast down in the tree. Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF-wave. Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that there is no process of status C in the tree and no other process can join it. So, the tree can be safely reset from the

Algorithm 1 Algorithm $\mathcal{LE}$ for every process p

Variables: $p.idR \in \mathbb{N}; p.par \in \mathcal{N}_p \cup \{p\}; p.level \in \mathbb{N}; p.status \in \{C, EB, EF\}$;

Macros:

$$\begin{aligned}
Children_p &\equiv \{q \in \mathcal{N}_p \mid q.par = p\}\\
RealChildren_p &\equiv \{q \in Children_p \mid KinshipOk(q,p)\}\\
p \preceq q &\equiv (p.idR \leq q.idR) \wedge [(p.idR = q.idR) \Rightarrow (p \leq q)]\\
Min_p &\equiv \min_{\preceq} \{q \in \mathcal{N}_p \mid q.status = C\}
\end{aligned}$$

Predicates:

$$\begin{aligned}
SelfRoot(p) &\equiv p.par = p\\
SelfRootOk(p) &\equiv (p.level = 0) \wedge (p.idR = p) \wedge (p.status = C)\\
GoodIdR(s,f) &\equiv (s.idR \geq f.idR) \wedge (s.idR < s)\\
GoodLevel(s,f) &\equiv (s.idR = f.idR) \Rightarrow (s.level = f.level + 1)\\
GoodStatus(s,f) &\equiv [(s.status = EB) \Rightarrow (f.status = EB)]\\
&\quad \vee[(s.status = EF) \Rightarrow (f.status \neq C)]\\
&\quad \vee[(s.status = C) \Rightarrow (f.status \neq EF)]\\
KinshipOk(s,f) &\equiv GoodIdR(s,f) \wedge GoodLevel(s,f) \wedge GoodStatus(s,f)\\
AbRoot(p) &\equiv [SelfRoot(p) \wedge \neg SelfRootOk(p)]\\
&\quad \vee[\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)]\\
Allowed(p) &\equiv \forall q \in Children_p, (\neg KinshipOk(q,p) \Rightarrow q.status \neq C)
\end{aligned}$$

Guards:

$$\begin{aligned}
EBroadcast(p) &\equiv (p.status = C) \wedge [AbRoot(p) \vee (p.par.status = EB)]\\
EFeedback(p) &\equiv (p.status = EB) \wedge (\forall q \in RealChildren_p, q.status = EF)\\
Reset(p) &\equiv (p.status = EF) \wedge AbRoot(p) \wedge Allowed(p)\\
Join(p) &\equiv (p.status = C) \wedge [\exists q \in \mathcal{N}_p, (q.idR < p.idR) \wedge (q.status = C)]\\
&\quad \wedge Allowed(p)
\end{aligned}$$

Actions:

$$\begin{aligned}
&EB\text{-}action :: EBroadcast(p) && \rightarrow p.status \leftarrow EB;\\
&EF\text{-}action :: EFeedback(p) && \rightarrow p.status \leftarrow EF;\\
&R\text{-}action :: Reset(p) && \rightarrow p.status \leftarrow C; p.par \leftarrow p;\\
& && \quad p.idR \leftarrow p; p.level \leftarrow 0;\\
&J\text{-}action :: Join(p) \wedge \neg EBroadcast(p) && \rightarrow p.par \leftarrow Min_p; p.idR \leftarrow p.par.idR;\\
& && \quad p.level \leftarrow p.par.level + 1;
\end{aligned}$$

abnormal root toward the leaves. Notice that the new variable $status$ may also get arbitrary initialization. Thus, we enforce previously introduced predicates as follows. A self root must have status C, otherwise it is an abnormal root:

$$SelfRootOk(p) \equiv SelfRootOk'(p) \wedge (p.status = C)$$

To be a real child of q, p should have a status coherent with the one of q. This is expressed with the predicate $GoodStatus(p,q)$ which is used to enforce the $KinshipOk(p,q)$ relation:

$$\begin{aligned}
GoodStatus(p,q) \equiv\ & [(p.status = EB) \Rightarrow (q.status = EB)]\\
& \vee[(p.status = EF) \Rightarrow (q.status \neq C)]\\
& \vee[(p.status = C) \Rightarrow (q.status \neq EF)]\\
KinshipOk(p,q) \equiv\ & KinshipOk'(p,q) \wedge GoodStatus(p,q)
\end{aligned}$$

Precisely, when p has status C, its parent must have status C or EB (if the EB-wave is not propagated yet to p). If p has status EB, then the status of its parent must be EB because p gets status EB from its parent q and q will change its status to EF only after p gets status EF. Finally, if p has status EF, its parent can have status EB (if the EF-wave is not propagated yet to its parent) or EF.

Normal Execution. Remark that, after all abnormal trees have been removed, all processes have status C and the algorithm works as in the initial version. Notice that the guard of $J\text{-}action$ has been enforced so that only processes with status C and which are not abnormal root can execute it, and when executing $J\text{-}action$, a process can only choose a neighbor of status C as parent. Moreover, remark that the cleaning of all abnormal trees does not ensure that all fake IDs have been removed. Rather, it guarantees the removal of all fake IDs smaller than ℓ. This implies that (at least) ℓ is a self root at the end of the cleaning and all other processes will elect ℓ within the next $\mathcal{D}$ rounds.

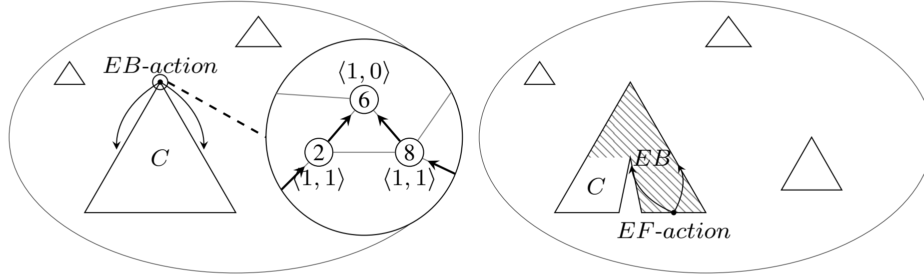
Cleaning Abnormal Trees. Figure 5 shows how an abnormal tree is cleaned. In the first phase (see Figure 5a), the root broadcasts status EB down to its (abnormal) tree: all the processes in this tree execute $EB\text{-}action$, switch to status EB and are consequently informed that they are in an abnormal tree. The second phase starts when the EB-wave reaches a leaf. Then, a convergecast wave of status EF is initiated thanks to action $EF\text{-}action$ (see Figure 5b). The system is asynchronous, hence all the processes along some branch can have status EF before the broadcast of the EB-wave is done into another branch. In this case, the parent of these two branches waits that all its children in the tree (processes in the set $RealChildren$) get status EF before executing $EF\text{-}action$ (Figure 5c). When the root gets status EF, all processes have status EF: the tree is dead. Then (third phase), the root can reset (safely) to become a self root by executing $R\text{-}action$ (Figure 5e). Its former real children (of status EF) become themselves abnormal roots of dead trees (Figure 5f) and reset.

Finally, we used the predicate $Allowed(p)$ to temporarily lock the parent of p in two particular situations — illustrated in Figure 6 — where p is enabled to switch its status from C to EB. These locks impact neither the correctness nor the complexity of $\mathcal{LE}$. Rather, they allow us to simplify the proofs by ensuring that, once enabled, $EB\text{-}action$ remains continuously enabled until executed.

4 Correctness and Complexity Analysis

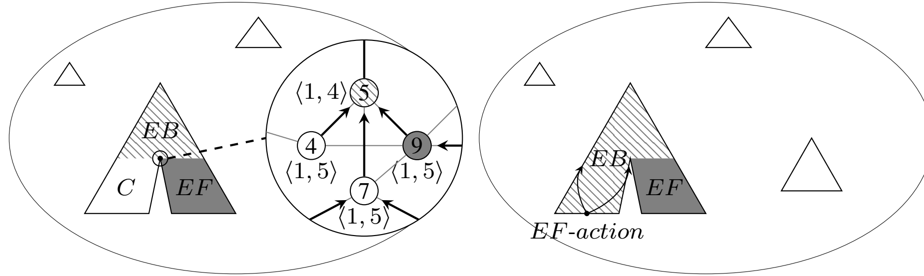
First, remark that idR and $level$ can be stored in $\Theta(\log n)$ bits. So, the memory requirement of $\mathcal{LE}$ is $\Theta(\log n)$ bits per process.

Let us first distinguish between *clean* and *dirty* configurations. Given any configuration γ, γ is *clean* if and only if in γ, $\forall p \in V, \neg EBroadcast(p) \wedge p.status = C$. In other words, a configuration is clean if and only if it contains no abnormal trees. In particular, such a clean configuration does not contain fake IDs smaller than ℓ. Any configuration that is not clean is said to be *dirty*.



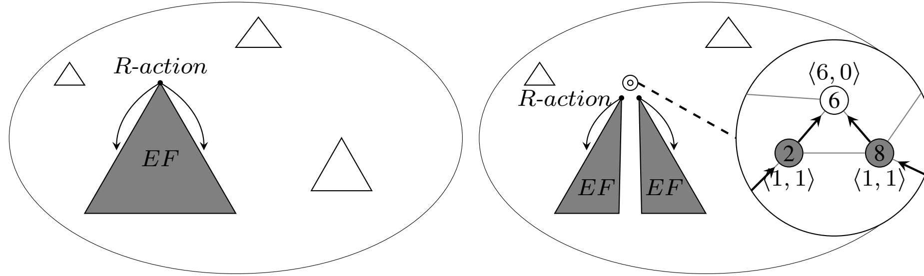
(a) When an abnormal root detects an error, it executes EB-$action$. The EB-wave is broadcast to the leaves. Here, 6 is an abnormal root because it is a self root and its idR is different from its ID ($1 \neq 6$).

(b) When the EB-wave reaches a leaf, it executes EF-$action$. The EF-wave is propagated up to the root.



(c) It may happen that the EF-wave reaches a node, here process 5, even though the EB-wave is still broadcasting into some of its proper subtrees: 5 must wait that the status of 4 and 7 become EF before executing EF-$action$.

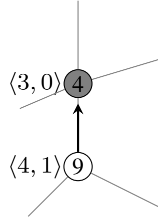
(d) EB-wave has been propagated in the other branch. An EF-wave is initiated by the leaves.



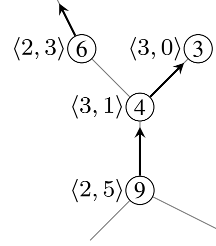
(e) EF-wave reaches the root. The root can safely reset (R-$action$) because its tree is dead. The cleaning wave is propagated down to the leaves.

(f) Its children become themselves abnormal roots of dead trees and can execute R-$action$: 2 and 8 can clean because their status is EF and their parent has status C.

Fig. 5: Schematic example of the cleaning mechanism. Trees are filled according to the status of their processes: white for C, dashed for EB, gray for EF.



(a) 4 and 9 are abnormal roots. If 4 executes R-$action$ before 9 executes EB-$action$, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, EB-$action$ is no more enabled at 9.



(b) 9 is an abnormal root and Min_4 is 6. If 4 executes J-$action$ before 9 executes EB-$action$, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, EB-$action$ is no more enabled at 9.

Fig. 6: Example of situations where the parent of a process is locked.

4.1 Correctness and Stabilization Time in Steps

Convergence from a clean configuration. Let us first consider any *clean* configuration, γ. As γ is clean, γ may contain some fake IDs, but all of them (if any) are greater than ℓ. This implies, in particular, that ℓ is a self root and $\ell.idR = \ell$ forever from γ. Moreover, in γ there are at most n different values disseminated into the idR variables. Every process $p \neq \ell$ can only decrease its own value of idR by executing J-$action$ (all other actions are disabled forever at p because they deal with abnormal trees). Hence, overall after at most $\frac{(n-1)\times(n-2)}{2}$ executions of J-$action$, the configuration is terminal and ℓ is elected.

Convergence from an arbitrary configuration. The remainder of the proof consists in showing that, from any arbitrary configuration, a clean configuration is reached in $O(n^3)$ steps. So, let consider a dirty configuration γ. Then, γ contains some abnormal trees. In the following, we say that a process p is called *alive* if and only if $p.status = C$. Otherwise, it is said to be *dead*. By extension, a tree T is called an *alive tree* if and only if $\exists p \in T$ such that p is alive. Otherwise, it is called a *dead tree*.

We first show that no abnormal alive tree can be created from γ. So, as there are at most n abnormal alive trees in the initial configuration, and each of them may contain up to n processes, at most n^2 EB-$action$, EF-$action$, and R-$action$ respectively are sufficient to freeze and remove all them. Notice that this way we clean abnormal trees is the main difference between our algorithm $\mathcal{LE}$ and the algorithm proposed in [10], $\mathcal{DLV}$. Indeed, we have shown that, contrary to $\mathcal{LE}$, the correction mechanism implemented in $\mathcal{DLV}$ can involve a non-polynomial number of correction actions (see [2]).

Nevertheless, processes can execute J-$action$ during the removal of abnormal trees. In particular, a process p can leave an abnormal alive tree T by ex-

ecuting $J\text{-}action$ to join another (normal or abnormal) tree. However, in this case the value of $p.idR$ necessarily decreases. Later p can join T again, but this may happen only if p executes actions $EB\text{-}action$, $EF\text{-}action$, and $R\text{-}action$ at least once in the meantime. This means that p participates to the removal of some abnormal tree. Thus, each time p joins T again, the number of abnormal trees decreases, *i.e.*, p can join and leave T at most $n-1$ times.

Thus, each process (n) can join each abnormal tree (at most n) at most $n-1$ times using $J\text{-}action$ which gives an overall number of $J\text{-}action$s in $O(n^3)$.

To sum up, starting from any configuration, a terminal configuration where ℓ is elected is reached in $O(n^3)$ steps. (We prove a tighter bound in [2].)

4.2 Stabilization Time in Rounds

Let us consider a clean configuration γ. Again, γ may contain some fake IDs, but all of them (if any) are greater than ℓ. This implies, in particular, that ℓ is a self root and $\ell.idR = \ell$ forever from γ. ℓ being the minimum value in idR variables, ℓ is propagated, from neighbors to neighbors, into the idR variables and the system reaches a terminal configuration in $O(\mathcal{D})$ rounds.

Consider now a dirty configuration γ. From γ, all abnormal trees are frozen and removed in parallel using three waves: (1) the broadcast of the value EB from the abnormal roots to the leaves, (2) a convergecast of the value EF from the leaves to the abnormal roots, and (3) finally, the cleaning is performed top-down. As the maximum height of a tree is n, each of these waves is done in at most n rounds. Overall, abnormal trees are removed in at most $3n$ rounds.

Hence, the stabilization time is at most $3n + \mathcal{D}$ rounds.

5 Conclusion

We proposed a silent self-stabilizing leader election algorithm, called $\mathcal{LE}$, for bidirectional connected identified networks of arbitrary topology. Starting from any arbitrary configuration, $\mathcal{LE}$ converges to a terminal configuration, where all processes know the ID of the leader, this latter being the process of minimum ID. Moreover, as in most of the solutions from the literature, a distributed spanning tree rooted at the leader is defined in the terminal configuration.

$\mathcal{LE}$ is written in the locally shared memory model. It assumes the distributed unfair daemon, the most general scheduling hypothesis of the model. Moreover, it requires no global knowledge on the network (such as an upper bound on the diameter or the number of processes, for example). $\mathcal{LE}$ is asymptotically optimal in space, as it requires $\Theta(\log n)$ bits per process, where n is the size of the network. We analyzed its stabilization time both in rounds and steps. We showed that $\mathcal{LE}$ stabilizes in at most $3n + \mathcal{D}$ rounds, where $\mathcal{D}$ is the diameter of the network. We have also proven in the technical report [2] that for every

$n \geq 4$, for every $\mathcal{D}, 2 \leq \mathcal{D} \leq n-2$, there is a network of n processes in which a possible execution exactly lasts this complexity.

Finally, we proved that $\mathcal{LE}$ achieves a stabilization time polynomial in steps. More precisely, we have shown in the technical report [2] that its stabilization time is at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps. Still in [2], we have shown that for every $n \geq 4$, there exists a network of n processes (and of diameter 2) in which a possible execution exactly lasts $\frac{n^3}{6} + \frac{3}{2}n^2 - \frac{8}{3}n + 2$ steps, establishing then that the worst case is in $\Theta(n^3)$.

Perspectives of this work deal with complexity issues. In [10], Datta *et al* showed that it is easy to implement a silent self-stabilizing leader election which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(D)$ rounds (where D is an upper bound on $\mathcal{D}$). Nevertheless, processes are assumed to *know* D. It is worth investigating whether it is possible to design an algorithm which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(\mathcal{D})$ rounds without using any global knowledge. We believe this problem remains difficult, even adding some fairness assumption.

References

1. Afek, Y., Bremler-Barr, A.: Self-Stabilizing Unidirectional Network Algorithms by Power Supply. Chicago J. Theor. Comput. Sci. 1998 (1998)
2. Altisen, K., Cournier, A., Devismes, S., Durand, A., Petit, F.: Self-Stabilizing Leader Election in Polynomial Steps. Tech. rep., CNRS (2014), hal.archives-ouvertes.fr/hal-00980798
3. Arora, A., Gouda, M.G.: Distributed Reset. IEEE Trans. Computers 43(9), 1026–1038 (1994)
4. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time Optimal Self-stabilizing Synchronization. In: STOC. pp. 652–661 (1993)
5. Blin, L., Tixeuil, S.: Brief Announcement: Deterministic Self-stabilizing Leader Election with O(log log n)-bits. In: PODC. pp. 125–127 (2013)
6. Burman, J., Kutten, S.: Time Optimal Asynchronous Self-stabilizing Spanning Tree. In: DISC. pp. 92–107 (2007)
7. Chang, E.J.H.: Echo Algorithms: Depth Parallel Operations on General Graphs. IEEE Trans. Software Eng. 8(4), 391–401 (1982)
8. Datta, A.K., Larmore, L.L., Piniganti, H.: Self-stabilizing Leader Election in Dynamic Networks. In: SSS. pp. 35–49 (2010)
9. Datta, A.K., Larmore, L.L., Vemula, P.: An O(n)-time Self-stabilizing Leader Election Algorithm. J. Parallel Distrib. Comput. 71(11), 1532–1544 (2011)
10. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler. Theor. Comput. Sci. 412(40), 5541–5561 (2011)
11. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM 17(11), 643–644 (1974)
12. Dolev, S., Gouda, M.G., Schneider, M.: Memory Requirements for Silent Stabilization. Acta Inf. 36(6), 447–462 (1999)
13. Dolev, S., Herman, T.: Superstabilizing Protocols for Dynamic Distributed Systems. Chicago J. Theor. Comput. Sci. 1997 (1997)
14. Kravchik, A., Kutten, S.: Time Optimal Synchronous Self Stabilizing Spanning Tree. In: DISC. pp. 91–105 (2013)