# Self-Stabilizing Weak Leader Election in Anonymous Trees using Constant Memory per Edge

AJOY K. DATTA  STEPHANE DEVISMES  LAWRENCE L. LARMORE
VINCENT VILLAIN

**Abstract**

We propose a deterministic silent self-stabilizing algorithm for the weak leader election problem in anonymous trees. Our algorithm is designed in the message passing model, and requires only $O(1)$ bits of memory per edge. It does not necessitate the *a priori* knowledge of any global parameter on the network. Finally, its stabilization time is at most $3\mathcal{D}^2 \times (X + 2I_{\max} + 2)$ time units, where $\mathcal{D}$ is the diameter of the network, $X$ is an upper bound on the time to execute some recurrent code by processes, and $I_{\max}$ is the maximal number of messages initially in a link.

**keywords:** Self-stabilization, weak leader election, anonymous tree, message passing, intermittent faults.

## 1 Introduction

We consider the deterministic self-stabilizing *weak leader election* problem in anonymous trees. Self-stabilization [1] is a versatile technique to withstand *any* finite number of transient faults in a distributed system. Indeed, a self-stabilizing algorithm is able to recover correct behavior in finite time without external (*e.g.*, human) intervention, regardless of the *arbitrary* initial configuration of the system (*i.e.*, regardless the initial state of the processes and messages initially in the links), and therefore, also after the occurrence of transient faults, provided that these faults do not alter the code of the processes.

A particular class of self-stabilizing algorithms is that of silent algorithms. A self-stabilizing algorithm is *silent* [2] if it converges to a global state where values of communication registers used by the algorithm remain fixed. Consequently, from such a global state information exchanged between processes is also fixed. Silent (self-stabilizing) algorithms are typically proposed to solve static problem such as leader election.

In distributed computing, the *leader election* problem consists in distinguishing a single process, so-called the leader, among the others. This problem is fundamental, as it is a basic component of solutions to a number of important problems such as spanning tree construction, implementation of broadcasting and convergecasting methods, *etc.* Now, following the results of Yamashita and Kameda [3], there are anonymous trees whose symmetricity is 2 (*e.g.*, a tree of just two processes) for which deterministic leader election is impossible. To circumvent this negative result, Datta *et al* [4] propose a weakened version of the problem, called *weak leader election*: elect at least one, but at most two processes, and in the latter case the two processes should be neighbors. The major interest of solving the weak leader election problem is to reduce the global problem of electing one leader among $n$ processes to the local problem of electing one leader among at most two neighbors. For example, if two neighbors are elected instead of only one process, then one of them can be probabilistically elected within constant expected time, just by making each of those two neighbors toss a coin and exchange their result until one obtains head and the other obtains tail. Notice that a very close variant of the weak leader election problem, called *general election problem (GEP)*, has been proposed in [5]. This problem consists in electing, if possible a process, otherwise if possible an edge. In this latter paper, non-self-stabilizing solutions to GEP in rings are studied, assuming that process identifiers are not necessarily unique.

The bulk of self-stabilizing literature uses the *shared memory model with composite atomicity* introduced by Dijkstra [1]. In this high-level model, any process is able to read the states of every neighbor and update its own state in a single atomic step. Designing self-stabilizing algorithm using lower level communication models such as asynchronous message passing is more challenging. In one of the weakest forms of the message passing model, messages in transit can be intermittently lost, duplicated, or reordered. It is especially important to consider such a low level model since Varghese and Jayaram [6] prove that simple process crashes and restarts, and unreliable communication channels, can drive protocols to arbitrary states. To our knowledge, very few work in the self-stabilizing literature consider such a weak model [7, 8].

## 1.1 Contribution

In this paper, we give a deterministic silent self-stabilizing algorithm for the weak leader election problem in an anonymous tree. Our algorithm is expressed in the message passing model under the following weak assumptions. No bound is assumed on the link capacity (although our algorithm also works assuming any positive bound on the link capacity). Moreover, the system can be subjected to both transient faults and intermittent faults[1]. Arbitrary transient faults are withstood due to the self-stabilizing nature of the algorithm. But, in addition, our algorithm also tolerates frequent message lost, duplication, and reordering.

Our algorithm requires only $O(1)$ bits of memory per edge. It does not necessitate the *a priori* knowledge of any global parameter on the network such as an upper bound on the diameter or the number of processes. Finally, it achieves a stabilization time independent of the number of processes in the tree. Namely, we prove that the stabilization time is at most $3\mathcal{D}^2 \times (X + 2I_{\max} + 2)$ time units, where $\mathcal{D}$ is the diameter of the network, $X$ is an upper bound on the time to execute some recurrent code by processes, and $I_{\max}$ is the maximal number of messages initially in a link.

## 1.2 Related Work

To the best of our knowledge, no self-stabilizing algorithm in a message passing model for the weak leader election problem has been given until now. The related papers presented in the paragraph below [9, 10, 11, 4, 12] consider the locally shared memory model with composite atomicity.

The weak leader problem has been introduced by Datta and Larmore in [4]. In that paper, they give a weak leader election algorithm for anonymous trees which is silent and self-stabilizing. The stabilization time of their algorithm is $O(\mathcal{D})$ rounds and $n.\mathcal{D}$ steps ($n$ is the number of processes). The memory requirement of their solution is $O(1)$ bits per process. They assume a distributed unfair daemon, the most general scheduling assumption of the locally shared memory model with composite atomicity. In the same paper, they also consider two related problems, finding centers and medians. Any tree contains at least one but at most two centers (resp. medians). Hence a solution to either problem is also a solution for the weak leader election, but the converse is false. They prove lower bounds for the space complexity of silent self-stabilizing algorithms for these two problems, namely $\Omega(\log \mathcal{D})$ and $\Omega(\log n)$ bits per process for the centers and the medians finding, respectively. They also propose two algorithms, one for each of the two problems, matching the lower bounds and achieving the same time complexity as their weak leader election algorithm. Another self-stabilizing algorithm for finding centers in anonymous trees can be found in [12]. In this paper, Datta *et al* circumvent the space complexity lower bound by proposing a self-stabilizing but non-silent algorithm. This algorithm has a memory requirement in $O(1)$ space per process. The stabilization time of their solution is $\mathcal{D}$ rounds, assuming the distributed unfair daemon. Other silent self-stabilizing solutions for the centers and medians finding are given in [9, 10, 11]. However, they assume strong scheduling hypothesis, *i.e.*, central and locally central daemon. Algorithms in [9, 10] assume processes *a priori* know an upper bound $N$ on the number of processes, and their memory requirement is in $\Theta(\log N)$ (no time complexity is given). In [11], the space complexity is $\Theta(\log \mathcal{D})$ for finding centers and $\Theta(\log n)$ for finding medians. The step complexity

[1]The main difference between transient and intermittent faults is their frequency. In the former case, the time between two periods of faults is large enough to allow the system to recover. In the latter case, a self-stabilizing algorithm should be able to converge despite the frequent occurrence of faults during its stabilization phase.

is $O(n^3 + n^2 c_h)$ for finding centers, and $O(n^3 c_s)$ for finding medians, where $c_h$, $c_s$ are the maximum initial values of certain variables in the algorithm.

More generally, several papers investigate the possibility of self-stabilization in message passing, *e.g.*, [13, 14, 15, 16, 17, 18, 19, 20, 7]. The crucial assumption for communication links is their *boundedness*, *i.e.*, whether or not processes are aware of the maximum number of messages that can be in transit in a particular link. Gouda and Multari [13] show that for a wide class of problems, including the alternating bit protocol (ABP), deterministic self-stabilization is impossible using bounded memory per process when link capacities are unbounded. They also present a self-stabilizing version of the ABP with unbounded links that uses unbounded memory per process. Afek and Brown [15] present a self-stabilizing ABP replacing unbounded process memory by an infinite sequence of random numbers. However, we are interested here in deterministic algorithms. Katz and Perry [14] derive a self-stabilizing ABP to construct a self-stabilizing snapshot protocol. In turn, the snapshot protocol allows to transform almost all non-self-stabilizing protocols into self-stabilizing ones, but at a prohibitive cost, *e.g.*, it uses unbounded memory per process, while we target here a space complexity constant per edge. Guaranteeing self-stabilization with bounded memory per process for most of specifications requires considering bounded capacity links and such a bound should be *a priori* known by all processes [16, 17, 18, 19, 20]. In particular, Varghese [18] gives such self-stabilizing solutions for a wide class of problems, including token circulation and propagation of information with feedback. In this paper, no bound is assumed on the link capacity.

Awerbuch *et al* [21] introduced the property of *local correctability* and demonstrated that protocols which are locally correctable can be self-stabilized using bounded memory per process in spite of unbounded capacity links. Delaët *et al* [7] also proposed a method to design silent self-stabilizing protocols with bounded memory per process in message passing systems equipped with unreliable links of unbounded capacity for a class of fix-point problems. However, solutions in [4, 12] are based on a local synchronization mechanism called *unison*, which cannot be implemented in message-passing using the methods proposed in [21, 7]. Finally, porting the other solutions for finding centers or medians [9, 10, 11] to message-passing using [21] or [7] (if possible) would give solutions with space complexity $\Omega(\Delta\mathcal{D})$ for finding centers and $\Omega(\Delta n)$ for finding medians (where $\Delta$ is the maximum degree of the tree), and so not in constant space per edge.

Finally, note that the design of the algorithm we propose is close to the one of the non-self-stabilizing election algorithm for identified tree networks proposed in the book of Gerard Tel [22] (page 231).

### 1.3 Roadmap

The next section is dedicated to computational model and basic definition. Our algorithm, $\mathcal{L}$, is presented in Section 3, while its correctness proof is given in Section 4. A complexity analysis of $\mathcal{L}$ is given in Section 5. We make concluding remarks in Section 6.

## 2 Preliminaries

### 2.1 Network

Let $T = (V, E)$ be an unoriented tree, where $V$ is a set of $n \geq 2$ deterministic processes, and $E$ a set of bidirectional asynchronous links between processes. Two processes $p$ and $q$ are said to be *neighbors* if $\{p, q\} \in E$. Each bidirectional link $\{p, q\} \in E$ can be decomposed into two unidirectional links $(p, q)$ and $(q, p)$ which are the links from $p$ to $q$ and from $q$ to $p$, respectively. Processes are *anonymous*. For any process $p$ and $q$ in $V$, we define $\|p, q\|$, the *distance* between $p$ and $q$ in $T$, to be the length of the shortest path in $T$ linking $p$ to $q$. The *diameter* $\mathcal{D}$ of $T$ is the maximum distance between any two processes, *i.e.*, $\mathcal{D} = \max_{p,q \in V} \|p, q\|$.

Each process can communicate with its neighbors by sending messages along the corresponding links. A link may hold an arbitrary number of messages. A process $p$ can distinguish the links it is incident to using distinct local labels. Let $\mathcal{N}_p$ be the set of all local labels at a process $p$, and let $\delta_p = |\mathcal{N}_p|$ be the *degree* of $p$. In the following, we will let $p$ denote both a process and the local label of $p$ at a neighbor $q$. Any process

$l$ whose degree is 1 is called a *leaf* and, in this case, we denote by $u(l)$ its unique neighbor. Let $Leaves_T$ be the set of leaves in $T$. For every leaf $l$, for every node $p \neq l$, let $c(l, p)$ be the neighbor of $p$ that is the closest from $l$.

## 2.2 States and Configurations

The *state* of a process is a vector of the values of its variables. Among the variables at process $p$, we make the distinction between communication and non-communication variables. A *communication variable* of process $p$ is a variable whose value is sent and/or received through messages. The state of a unidirectional link $(p, q)$ is a multiset of messages[2] $M_{p,q}$. The state of the bidirectional link $\{p, q\}$ is the unordered pair $\{M_{p,q}, M_{q,p}\}$. When process $p$ sends a message $m$ to process $q$, $m$ is added to $M_{p,q}$; eventually $m$ is deleted from that set when $m$ is either lost, or received by $q$. A configuration of the system is the product of the states of the processes and the states of the bidirectional links. Each configuration contains finitely many messages.

## 2.3 Events

The system configuration is modified by *atomically* executing an *applicable event*: if $\gamma$ is a configuration and $e$ an event applicable to $\gamma$, we denote by $e(\gamma)$ the configuration obtained by atomically executing $e$ on $\gamma$.

There are two kinds of events: *process events* and *system events*.

### 2.3.1 Process Events

A *process event* is executed by some process $p$, where $p$ can modify its state and send finitely many messages. There are two kinds of process events: *recurring events* ($R$ event) and *triggered events* ($T$ event).

- A recurring event at process $p$ is regularly executed by $p$, perhaps by using a timer. We make no assumption about the periodicity of the event, except that it is executed infinitely often. By definition, a recurring process event is applicable to any configuration.
- A triggered event can only be executed by a process $p$ if it is *triggered* by the reception of an expected message $m$ from a neighbor $q$, in which case $m$ is removed from $M_{q,p}$. Of course, the event is applicable only if $m \in M_{q,p}$.

### 2.3.2 System Events

A system event is triggered by an intermittent fault. We assume two kinds of intermittent faults can occur: message loss and message duplication. A message loss consists of removing some message $m$ from some multiset $M_{p,q}$. A message duplication consists in adding an additional copy of some message $m$ in some multiset $M_{p,q}$. Either of these events is applicable only if $m \in M_{p,q}$.

## 2.4 Local and Distributed Algorithms

A distributed algorithm consists of $n$ local algorithms, one per process. The local algorithm of $p$ consists of definitions of finitely many process events.

## 2.5 Schedule and Execution

A *schedule* is a sequence of events. A schedule $S = e_0, \dots$ is *admissible* from some configuration $\gamma_0$ if

- $e_0$ is applicable to $\gamma_0$ and $\forall i > 0$, if $e_i$ is defined, then $e_i$ is applicable to $\gamma_i$ where $\gamma_i = e_{i-1}(\gamma_{i-1})$, and
- the following well-foundedness and fairness properties are satisfied:

[2]This assumption implies that the links are not FIFO.

  - Every event of type $R$ at every process appears infinitely often in $S$. (Process fairness.)
  - Every message $m$ in a link $(q, p)$ that is not lost is eventually received, provided that there is an appropriate $T$ event in the local algorithm of $p$, *i.e.*, a $T$ event in the local algorithm of $p$ that is triggered by the reception of $m$ from $q$. (Eventual delivery property.)
  - If infinitely many messages are sent in a link $(q, p)$, then infinitely many messages are received by $p$, provided that there are the appropriate $T$ events in the local algorithm of $p$. (The links are fair lossy.)
  - Each message is duplicated only a finite (yet unbounded) number of times and only if it is a message that has been previously sent, or a message that is present in the initial configuration. (Finite number of duplications.)

Let $\gamma_0$ be a configuration and $S = e_0, \ldots$ an admissible schedule from $\gamma_0$. The corresponding *execution* is $\gamma_0, \gamma_1 = e_0(\gamma_0), \ldots$

### 2.6 Silent Self-Stabilization

*Silent self-Stabilization* is introduced in [2] as a specialization of *self-stabilization* [1]. A configuration $\gamma$ is *terminal* if the values of all communication variables are constant in all possible executions starting from $\gamma$.

Let $SP$ be a predicate over configurations. An algorithm is *silent and self-stabilizing for $SP$* in a given network, if all its executions reach within a finite time a terminal configuration from which $SP$ holds forever in any possible execution suffix.

### 2.7 The Weak Leader Election Problem

The *weak leader election problem* consists in making each process $p$ computes the value of some local predicate $Leader(p)$ so that the predicate $\mathcal{S}_{WL}$ defined below is eventually satisfied forever.

For any configuration $\gamma$, we say that $\mathcal{S}_{WL}(\gamma)$ holds if

1. there is at least one but at most two processes satisfying predicate $Leader$ in $\gamma$; and
2. if there are two processes satisfying predicate $Leader$ in $\gamma$, they are neighbors.

Hence, regardless the initial configuration of the system, a silent self-stabilizing algorithm for the weak leader election problem should reach within finite time a terminal configuration from which

1. the value of $Leader(p)$ is constant for every process $p$ and
2. $\mathcal{S}_{WL}$ holds forever in any possible execution suffix.

## 3 Algorithm

Our algorithm is called Algorithm $\mathcal{L}$. Its formal code is given in Algorithm 1. Algorithm $\mathcal{L}$ aims at electing one process or one edge. In the former case the elected process is called the *leader*. In the latter case, the two processes incident to the elected edge are called *co-leaders*. We now define $\mathcal{L}$.

### 3.1 Variables

Each process $p$ maintains two communication variables:

- $P_p \in \mathcal{N}_p \cup \{\bot\}$, which we refer to as the *pointer* of $p$.

  $P_p = \bot$ means that $p$ is currently candidate for the leader election. Otherwise, $P_p = q \in \mathcal{N}_p$ and we have two cases. Either $P_q = p$ and the edge $\{p, q\}$ is a leader edge, or $p$ is not candidate and adopts the same leader as $q$: $q$ is on the shortest path to the (co-)leader process(es).

- $A_p[]$, a Boolean array of size $\delta_p$ indexed by $\mathcal{N}_p$. $A_p[q]$ is true if and only if $p$ believes that $q$ points at $p$, *i.e.*, that $P_q = p$.

In a terminal configuration of $\mathcal{L}$, either the pointers define a tree rooted at the (sole) leader, or two trees rooted at the co-leaders and the two co-leaders point to each other.

The memory requirement of each process $p$ is $\delta_p + \lceil \log(\delta_p + 1) \rceil$ bits, *i.e.*, $O(\Delta)$ bits where $\Delta = \max_{p \in V} \delta_p$ is the maximum degree of the tree, *i.e.*, $O(1)$ per edge. So, we have

**Theorem 1** *The memory requirement in Algorithm $\mathcal{L}$ is constant per edge.*

Finally, as there are $n-1$ edges in a tree, the average memory requirement per process is also constant.

## 3.2 Messages

$\mathcal{L}$ has two possible messages: $\langle 0 \rangle$ and $\langle 1 \rangle$. If $p$ points to a neighbor $q$, then it continually sends the message $\langle 1 \rangle$ to $q$; otherwise it continually sends $\langle 0 \rangle$ to $q$. Precisely, each process $p$ regularly sends a one-bit message $\langle P_p = q \rangle$ to each neighbor $q$, where $\langle P_p = q \rangle$ equals $\langle 1 \rangle$ if $P_p = q$, 0 otherwise.

For any neighboring processes $p$ and $q$, $A_p[q] \leftarrow 1$ whenever $p$ receives the message $\langle 1 \rangle$ from $q$, and $A_p[q] \leftarrow 0$ whenever $p$ receives the message $\langle 0 \rangle$ from $q$.

## 3.3 Predicate *Leader*

The specification predicate of $\mathcal{L}$ is instantiated as follows:

$$Leader(p) \equiv (P_p = \perp) \vee (A_p[P_p] = 1)$$

Algorithm $\mathcal{L}$ ensures that in any terminal configuration, $(P_p = q \in \mathcal{N}_p) \Leftrightarrow (A_q[p] = 1)$, for every process $p$ and every $q \in \mathcal{N}_p$. There are two kinds of terminal configuration. In the first kind, $P_p = \perp$ for exactly one process $p$ and there is no elected edge. In this case, $p$ is the sole leader, *i.e.*, the unique process $p$ satisfying $Leader(p)$. Otherwise, no process has its pointer equal to $\perp$ and exactly one edge is elected. The ends of that edge, say $p$ and $q$, point to each other, that is, $P_p = q$ and $P_q = p$, which implies both $Leader(p)$ and $Leader(q)$, and no other process satisfies the specification predicate.

## 3.4 Macro and Function

Algorithm $\mathcal{L}$ uses the following macro:

$$N0(p) = \{q \in \mathcal{N}_p : A_p[q] = 0\}$$

Then, $\mathcal{L}$ uses the function $NewP(p)$, defined below, to recompute the value of the pointer $P_p$ continually. Eventually, all pointer values are stable, *i.e.*, do not change, and $\mathcal{L}$ has converged.

```
NewP(p)
1:   if        |N0(p)| = 1                               then
2:                 return the unique element of N0(p)
3:   else if   |N0(p)| ≥ 2                               then
4:                 return ⊥
5:   else
6:                 return P_p
```

**Algorithm 1** Algorithm $\mathcal{L}$, code for any process $p$

```
R event:
1:    $P_p \leftarrow NewP(p)$
2:    for all $q \in \mathcal{N}_p$ do send$\langle P_p = q \rangle$ to $q$
T event: upon receiving $\langle x \rangle$ from $q$
3:    $A_p[q] \leftarrow x$
4:    $P_p \leftarrow NewP(p)$
```

### 3.5 Overview

Every process $p$ regularly sends to each neighbor $q$ a message ($\langle 0 \rangle$ or $\langle 1 \rangle$) to let it know whether $P_p = q$ (see Line 2 in the $R$ event of Algorithm 1). Upon receiving this message from $p$, $q$ updates $A_q[p]$ accordingly (see Line 3 in the $T$ event of Algorithm 1).

If $P_p = \perp$, then $p$ is a candidate for leader election. If $P_p = q \neq \perp$ there are two cases. If $P_q = p$, then the edge $\{p, q\}$ is elected, meaning that both $p$ and $q$ consider themselves to be co-leaders. Otherwise, $p$ is no longer a candidate for leader election, and $q$ is the neighbor of $p$ on the shortest path to the (co-)leader(s).

The election process works as follows. Each process $p$ updates $P_p$ using function $NewP(p)$. $P_p$ is regularly updated (see Line 1 in the $R$ event of Algorithm 1) and also after each update of $A_p[]$ (see Line 4 in the $T$ event of Algorithm 1):

- $p$ decides to point to a neighbor $q$ if all its neighbors except $q$ point to $p$, *i.e.*, $N0(p) = \{q\}$ (see Lines 1-2 in $NewP(p)$): $p$ sets $P_p$ to $q$ meaning that it is no longer a candidate, and adopts the same leader process or the same leader edge as $q$. Notice that it is possible that $p$ and $q$ decide to point each other concurrently. In this case, the edge $\{p, q\}$ is a leader edge.
- $p$ sets $P_p$ to $\perp$ if $|N0(p)| \geq 2$ (see Lines 3-4 in $NewP(p)$), because there is currently no agreement between at least two of its neighbors on where the leader is.
- In all other cases, $P_p$ is not modified: see Lines 5-6 in $NewP(p)$.

## 4 Correctness

Recall that we consider trees of $n$ nodes with $n \geq 2$.

**Lemma 1** *Let $p$ and $q$ be two neighboring processes, If, within finite time, the system reaches a configuration $\gamma$ from which $P_p = q$ (resp. $P_p \neq q$) forever, then within finite time (from $\gamma$), we have*

1. *$q$ only receives messages $\langle x \rangle$ from $p$,*
2. *$q$ receives messages $\langle x \rangle$ from $p$ infinitely often, and*
3. *$A_q[p] = x$ forever,*

*where $x = 1$ if $P_p = q$ in $\gamma$, $x = 0$ otherwise.*

**Proof.** The proof of this lemma is immediate from the definition of the algorithm and the assumptions on the system, as shown below.

- *Proof of Claim 1:* The number of messages initially in the link $(p, q)$ is finite (by definition, each configuration contains only a finite number of messages). Then, by definition of the algorithm, from $\gamma$, $p$ only sends messages $\langle x \rangle$ to $q$. Finally, each message can be duplicated only a finite number of times and each message is eventually received or lost (see the properties of an admissible schedule). Hence, eventually $q$ can only receive messages $\langle x \rangle$ from $p$.

- *Proof of Claim 2:* By definition, $p$ executes its $R$ event infinitely often. So, Claim 2 is a consequence of the initial hypothesis of the lemma and the fact that links are fair lossy.

- *Proof of Claim 3:* Immediate from Claims 1 and 2.

$\square$

By definition, a leaf $l$ cannot execute Lines 3-4 of $NewP(l)$ because $\delta_l = 1$. Hence follows.

**Remark 1** *For every leaf $l$, if $P_l = u(l)$, then $P_l = u(l)$ forever.*

**Lemma 2** *Let $l$ be a leaf. Within finite time, the system reaches a configuration $\gamma$ from which*

1. *$P_l$ is constant forever.*

*Moreover, let $x = 1$ if $P_l = u(l)$ in $\gamma$, $x = 0$ otherwise. Within finite time (from $\gamma$), we have*

2. *$u(l)$ only receives messages $\langle x \rangle$ from $l$,*

3. *$u(l)$ receives messages $\langle x \rangle$ from $l$ infinitely often, and*

4. *$A_{u(l)}[l] = x$ forever.*

**Proof.**

- *Proof of Claim 1:* By the contradiction. Then, $P_l$ switches from $u(l)$ to $\perp$ infinitely often, a contradiction to Remark 1.

- *Proof of Claims 2-4:* Immediate from Claim 1 and Lemma 1.

$\square$

Then, the proof of correctness is done by induction on the number $n$ of processes in the tree.

## 4.1 Base case

If the tree contains only two processes $p$ and $q$, both are leaves and neighbors. By Lemma 2 (Claims 1 and 4), the system eventually reaches a terminal configuration $\gamma$. Assume, by the contradiction, that neither $Leader(p)$ nor $Leader(q)$ hold in $\gamma$. In this case, $P_p = q \wedge A_p[q] = 0$ and $P_q = p \wedge A_q[p] = 0$. Now, in $\gamma$, $P_p = q$ implies that $A_q[p] = 1$ and $P_q = p$ implies that $A_p[q] = 1$ (by Lemma 2.4), a contradiction. Hence, $\mathcal{S}_{WL}(\gamma)$ holds, and we are done.

## 4.2 Inductive Hypothesis

Let $k \geq 2$. Assume that every execution of $\mathcal{L}$ in any tree $T$ of $n$ nodes, with $2 \leq n \leq k$, reaches within finite time a terminal configuration $\gamma$ from which $\mathcal{S}_{WL}$ holds forever.

## 4.3 Inductive Step

Let $T = (V, E)$ be a tree of $k + 1$ processes. Let $e = \gamma_0 \dots$ be any execution of $\mathcal{L}$ on $T$. We consider the following two cases.

### 4.3.1 Case 1

Assume that there is a leaf $l$ such that, within finite time in $e$, $P_{u(l)} \neq l$ holds forever. Then, the execution $e$ eventually reaches a configuration $\gamma$ from which $A_l[u(l)] = 0$ forever, by Lemma 1. After its first $R$ event from $\gamma$, $l$ satisfies $P_l = u(l)$ forever, by Remark 1. After that, the execution $e$ eventually reaches a configuration from which $A_{u(l)}[l] = 1$ forever by Lemma 2.4. So:

- Eventually in $e$, the state of $l$ is constant and $Leader(l)$ is false forever. (Indeed, $P_l = u(l) \wedge A_l[u(l)] = 0$ holds forever.)
- Moreover, once $A_{u(l)}[l] = 1$ forever in $e$, $l$ is neutral in the behavior of $u(l)$ and in particular in the computation of $P_{u(l)}$ since $l$ does not belong to $N0(u(l))$ anymore. More precisely, let $T' = (V \setminus \{l\}, E \setminus \{l, u(l)\})$ and let $\gamma_i$ be the first configuration of $e$ from which $A_{u(l)}[l] = 1$ forever in $e$. Let $\gamma'_i$ be the configuration of $T'$ obtained by removing from $\gamma_i$

  - the state of $l$,
  - the state of the link $\{l, u(l)\}$, and
  - the array cell $A_{u(l)}[l]$ in the state of $u(l)$.

  Let $S_e = e_0, \ldots$ be the schedule that generates $e$. Let $S_{e'} = e_a, e_b, \ldots$ be the schedule obtained by removing from $S_e$ all events related to $l$ (*i.e.* process events at $l$, messages received by $u(l)$ from $l$, messages received by $l$ from $u(l)$, and system events in the link $\{l, u(l)\}$). $S_{e'}$ is admissible from $\gamma'_i$. Let $e' = \gamma'_i, \gamma'_a = e_a(\gamma'_i), \gamma'_b = e_b(\gamma'_a), \ldots$ be the corresponding execution. For every configuration $\gamma'_z$ in $e'$, $\forall v \in V \setminus \{l\}$, $\forall w \in \mathcal{N}_v \setminus \{l\}$, we have

  - the state of $v$ in $\gamma'_z$ is equal to the state of $v$ in $\gamma_z$, except for $A_{u(l)}[l]$ which does not exist in $\gamma'_z$ while $A_{u(l)}[l] = 1$ in $\gamma_z$; and
  - the state of the link $\{v, w\}$ is the same in $\gamma_z$ and $\gamma'_z$.

  By the inductive hypothesis, $e'$ stabilizes to a terminal configuration from which $\mathcal{S}_{WL}$ holds forever. So, $e$ stabilizes to a terminal configuration from which $\mathcal{S}_{WL}$ holds forever too: there are at least one but at most two leaders among processes in $V \setminus \{l\}$, and $l$ is not a leader.

### 4.3.2 Case 2

Assume that for every leaf $l$, we have $P_{u(l)} = l$ infinitely often. Consider now a particular leaf $l$. We can show, by induction on the distance of processes $p \neq l$ to $l$ that $P_p = c(l, p)$ infinitely often.

- *Base Case:* The statement holds trivially for $u(l)$, by hypothesis.
- *Inductive Hypothesis:* Let $d \geq 1$. Assume that for every process $p$ at distance $d$ from $l$, we have $P_p = c(l, p)$ infinitely often.
- *Inductive Step:* let $p$ be a process at distance $d + 1$ from $l$. Let $q = c(l, p)$. $q$ is at distance $d$ from $l$. By the inductive hypothesis, $P_q = c(l, q) \neq p$ infinitely often. Now, each time $q$ sets $P_q$ to $c(l, q)$, we have $N0(q) \leq 1$:

  - either $q$ executes Line 2 in Function $NewP(q)$ and $N0(q) = \{c(l, q)\}$, or
  - $q$ executes Line 6 in Function $NewP(q)$ and $N0(q) = \emptyset$.

  In either case, $A_q[p] = 1$. Thus, $A_q[p] = 1$ infinitely often. Now, $q$ receives infinitely many messages from $p$. So, $q$ receives infinitely many messages $\langle 1 \rangle$ from $p$. Consequently, $p$ satisfies $P_p = q = c(l, p)$ infinitely often.

We now use the previous result to obtain a contradiction. Let $v$ be a leaf different from $l$. $v$ exists since the number of nodes in $T$, $k+1$, is greater than of equal to 3. Again, since $k+1 \geq 3$, $u(v) \neq l$. We can then apply the previous result: $P_{u(v)} = c(l, u(v)) \neq v$ infinitely often. Now, by hypothesis, $P_{u(v)} = v$ infinitely often too. So, this means that $A_{u(v)}[v]$ oscillates between 0 and 1 infinitely often, a contradiction to Lemma 2.4.

Hence, the following theorem follows:

**Theorem 2** *Algorithm $\mathcal{L}$ is silent and self-stabilizing for $\mathcal{S}_{WL}$ in any anonymous tree of at least two processes.*

## 5 Stabilization Time and Message complexity

### 5.1 Measuring Complexity

The *stabilization time* of a silent self-stabilizing algorithm is the maximum time, starting from any arbitrary initial configuration and over all possible executions, to reach a terminal configuration from which the specification is true forever.

The problem is now how to measure complexities in asynchronous unreliable message passing self-stabilizing systems. In our model, messages can be lost. We make no assumption on the drop rate of messages and no assumption on the frequency of message sending. Time (resp. message) complexity cannot be bounded under these assumptions. The problem is similar with duplication of messages. To circumvent this problem, we only measure complexities in executions where all links are reliable (*i.e.* no message loss and no duplication). We justify this simplification by the fact that measuring time (resp. message) complexity under unreliable links does not only evaluate the performance of the algorithm, but rather the performance of the whole system, including the network. Following the literature (*e.g.*, [22]), we do not want to make complexity analysis dependent on system-specific parameters.

Under the reliable link hypothesis, we evaluate time complexity in terms of *time units*, assuming that in at most one time unit at least an oldest message in each link[3] (if any) reaches its destination, and that the time for executing the code of a process event is zero.

Now, one problem remains: what are the frequency of recurrent events at processes? If we assume that, like message transmissions, each recurrent event is executed at the latest every unit of time, then the links may suffer from congestion, *i.e.*, messages may be sent more often than they are received causing the number of messages gradually increasing in the network. So, to prevent any congestion, we assume that each recurrent event is executed at most once per time unit. Moreover, notice that assuming a lower bound on the time before executing a recurrent event is necessary to permit the message complexities to be bounded. Finally, to permit bounded time complexities, we assume that each recurrent event is executed at least every $X \geq 1$ time units. Informally, this means that the periodicity of the timers of recurrent events should be carefully addressed when deploying the algorithm. In practice, $X$ represents the ratio between the actual periodicity of the timers and the worst-case message transmission time. So, $X$ should be greater than or equal to 1 to prevent congestion. In the best case, the periodicity of the timers is chosen as small as possible, while preventing congestion ($X = 1$): in this case, timers have no impact on performances. Otherwise, the penalty is of a factor $X$.

### 5.2 Stabilization Time of Algorithm $\mathcal{L}$

Consider now an arbitrary execution $e$ where all links are reliable. Let $I_{\max}$ be the maximum number of messages in a link in the initial configuration of $e$. From our hypotheses, we deduce that are always at most $I_{\max}+1$ messages in a link. Hence, the next remark follows.

**Remark 2** *During $e$, any message is received within at most $I_{\max}+1$ time units after its appearance in a link.*

[3]We *partially order* messages in any execution $e = \gamma_0, \ldots$ using the index of the configuration where a message appears first.

**Lemma 3** *Let $l$ be any leaf. If $P_l = u(l)$, then within $X + 2I_{\max} + 2$ time units, the following conditions hold forever:*

- *$(l, u(l))$ only contains messages $\langle 1 \rangle$,*
- *$A_{u(l)}[l] = 1$, and*
- *the last message $u(l)$ received from $l$ is $\langle 1 \rangle$.*

**Proof.** Once $P_l = u(l)$, $P_l = u(l)$ forever, by Remark 1. Hence, from that time, $l$ only sends messages $\langle 1 \rangle$ to $u(l)$. So, after at most $I_{\max}+1$ time units, $(l, u(l))$ only contains messages $\langle 1 \rangle$ forever. Within $X+I_{\max}+1$ additional time units, a message $\langle 1 \rangle$ is necessarily sent by $l$ and received by $u(l)$. Hence, from that time, $A_{u(l)}[l] = 1$ forever and the last message $u(l)$ receives from $l$ is always $\langle 1 \rangle$. $\Box$

Following a similar reasoning, we have:

**Lemma 4** *Let $l$ be any leaf. Let $t \geq 0$ and $t' \geq t + X + 2I_{\max} + 2$, If $P_l = \perp$ at any time during $[t, t']$, then at any time in $[t + X + 2I_{\max} + 2, t']$ we have:*

- *$(l, u(l))$ only contains messages $\langle 0 \rangle$,*
- *$A_{u(l)}[l] = 0$,*
- *the last message $u(l)$ received from $l$ is $\langle 0 \rangle$.*

**Proof.** In $[t, t']$, $l$ only sends messages $\langle 0 \rangle$ to $u(l)$. So, after at most $I_{\max} + 1$ time units from $t$, $(l, u(l))$ only contains messages $\langle 0 \rangle$ until (at least) time $t'$. Within $X + I_{\max} + 1$ additional time units, a message $\langle 0 \rangle$ is necessarily sent by $l$ and received by $u(l)$. Hence, from that time and until (at least) $l$ modifies the value of $P_l$, $A_{u(l)}[l] = 0$ and the last message $u(l)$ receives from $l$ is always $\langle 0 \rangle$. $\Box$

**Lemma 5** *Let $l$ be any leaf. Let $t \geq 0$ and $t' \geq t + 2\mathcal{D} \times (X + 2I_{\max} + 2)$. If $P_{u(l)}$ is assigned to a value in $\{l, \perp\}$ each time $u(l)$ executes an $R$ event in $[t, t']$, then the configuration at time $t'$ is terminal.*

**Proof.**

- *Claim 1:* Let $t'' \geq t + (\mathcal{D} - 1) \times (X + 2I_{\max} + 2)$. If $P_{u(l)}$ is assigned to a value in $\{l, \perp\}$ each time $u(l)$ executes an $R$ event in $[t, t'']$, then for every node $v$ in $V \setminus \{l, u(l)\}$ and for every time in $[t + \|v, u(l)\| \times (X + 2I_{\max} + 2), t'']$ we have

  1. $A_v[c(l, v)] = 0$,
  2. if $v$ executes an $R$ event, then $P_v$ is assigned to a value in $\{c(l, v), \perp\}$,
  3. $(c(l, v), v)$ only contains messages $\langle 0 \rangle$, and
  4. the last message $v$ received from $c(l, v)$ is $\langle 0 \rangle$.

  *Proof of Claim 1:* By induction on the distance of $v$ to $u(l)$. The case $\|v, u(l)\| = 0$ is vacuum since, by definition, there is no node in $V \setminus \{l, u(l)\}$ at distance 0 from $u(l)$. Assume now, that $\|v, u(l)\| = k$ with $k > 0$. Assume that $v \neq l$ (the case $v = l$ is also vacuum). Notice that $\mathcal{D} > 1$ in this case. From time $t + (\|v, u(l)\| - 1) \times (X + 2I_{\max} + 2)$ to $t''$, $c(l, v)$ only sends messages $\langle 0 \rangle$ to $v$ (if $c(l, v) \neq u(l)$, we apply the induction hypothesis on $c(l, v)$, otherwise $c(l, v) = u(l)$, *i.e.*, $\|v, u(l)\| = 1$, and this fact is immediate from the initial hypothesis on $P_{u(l)}$, since $v \neq l$). Hence, from time $t + (\|v, u(l)\| - 1) \times (X + 2I_{\max} + 2) + I_{\max} + 1$ to $t''$, $(c(l, v), v)$ only contains messages $\langle 0 \rangle$. Within $X + I_{\max} + 1$ additional time units, we have the guarantee that $v$ received a message $\langle 0 \rangle$ from $c(l, v)$. Hence, from $t + \|v, u(l)\| \times (X + 2I_{\max} + 2)$ to $t''$, Conditions 1-4 hold.

- *Claim 2:* Let $t''' \geq t + (2\mathcal{D} - 2) \times (X + 2I_{\max} + 2)$. If $P_{u(l)}$ is assigned to a value in $\{l, \perp\}$ each time $u(l)$ executes an $R$ event in $[t, t''']$, then for every node $v \notin \{l, u(l)\}$ and for every time in $[t + (2\mathcal{D} - \|v, u(l)\| - 1) \times (X + 2I_{\max} + 2), t''']$ we have

  1. $P_v = c(l, v)$,
  2. $(v, c(l, v))$ only contains messages $\langle 1 \rangle$,
  3. the last message $c(l, v)$ received from $v$ is $\langle 1 \rangle$, and
  4. $A_{c(l,v)}[v] = 1$.

  *Proof of Claim 2:* By structural induction from the leaves different from $l$. Let $f \neq l$ be a leaf. By Claim 1.4 and Lemma 3, we are done. Let $v$ be a non-leaf process. By applying the induction hypothesis on every node $y$ of $\mathcal{N}_v \setminus \{c(l, v)\}$ and Claim 1, we obtain that Condition 1 holds from time $t + (2\mathcal{D} - \|v, u(l)\| - 2) \times (X + 2I_{\max} + 2)$ to $t'''$. Then, Conditions 2-4 are immediate from 1 and the fact that $X + 2I_{\max} + 2$ additional time units are necessary to flush the link $(v, c(l, v))$ and to set $A_{c(l,v)}[v]$ to 1.

Consider now the time interval from $t + (2\mathcal{D} - 2) \times (X + 2I_{\max} + 2)$ to $t + (2\mathcal{D} - 1) \times (X + 2I_{\max} + 2)$. If $P_l = \perp$ at every time of this interval, then the configuration at time $t + (2\mathcal{D} - 1) \times (X + 2I_{\max} + 2)$ is terminal, by Lemma 4 and Claims 1-2. Otherwise, within $2\mathcal{D} \times (X + 2I_{\max} + 2)$ time units from $t$ at the latest, the configuration is terminal, by Lemma 3 and Claims 1-2. □

**Theorem 3** *The stabilization time of Algorithm $\mathcal{L}$ is at most $3\mathcal{D}^2 \times (X + 2I_{\max} + 2)$ time units.*

**Proof.** By induction on the diameter $\mathcal{D}$ of the tree.

If $\mathcal{D} = 1$, then the tree consists of only two neighboring leaves $l_1$ and $l_2$. Without the loss of generality, consider $l_1$. $\mathcal{N}_{l_1} \setminus \{l_2\} = \emptyset$. Hence, the value of $P_{l_1}$ is always in $\{l_2, \perp\}$ and, by Lemma 5, the system reaches a terminal configuration in at most $2 \times (X + 2I_{\max} + 2)$ time units.

If $\mathcal{D} = 2$, then let $p$ be the unique non-leaf process. If $P_p$ is assigned to a given leaf $l$ or $\perp$ each time $p$ executes an $R$ event during the time interval $[0, 4 \times (X + 2I_{\max} + 2)]$, then within $4 \times (X + 2I_{\max} + 2)$ time units the configuration is terminal, by Lemma 5. Otherwise, there are at least two $R$ events in the time interval $[0, 4 \times (X + 2I_{\max} + 2)]$, where $p$ points to two different leaves. As a consequence, $p$ sends at least one message $\langle 0 \rangle$ to all leaves during this interval. Let $t = 4 \times (X + 2I_{\max} + 2)$. Within at most $I_{\max} + 1$ time units from $t$, all leaves receives a $\langle 0 \rangle$ message. Consequently, a time $t' = 4X + 9I_{\max} + 9$, every leaf $l$ satisfies $P_l = p$ forever (Remark 1). Apply Lemma 3 on all leaves. A particular consequence is that $P_p$ becomes constant from time $t'' = 5X + 11I_{\max} + 11$. After $X + 2I_{\max} + 2$ time units from $t''$, the configuration is terminal, *i.e.*, the system reaches a terminal configuration within at most $6X + 13I_{\max} + 13$ time units.

Assume $\mathcal{D} > 2$. Consider the two following cases:

- There is a leaf $l$ such that $P_{u(l)}$ is assigned to a value in $\{l, \perp\}$ each time $u(l)$ executes an $R$ event during the time interval $[0, 2\mathcal{D} \times (X + 2I_{\max} + 2)]$. Then, at time $2\mathcal{D} \times (X + 2I_{\max} + 2)$ the system is in a terminal configuration, by Lemma 5.
- Otherwise, all leaves receives a message $\langle 0 \rangle$ within time $t = 2\mathcal{D} \times (X + 2I_{\max} + 2) + I_{\max} + 1$. From time $t$, every leaf $l$ satisfies $P_l = u(l)$ forever. Moreover, by Lemma 3, from time $t' = (2\mathcal{D} + 1) \times (X + 2I_{\max} + 2) + I_{\max} + 1$, for every leaf $l$, we have
  - $(l, u(l))$ only contains messages $\langle 1 \rangle$,
  - $A_{u(l)}[l] = 1$, and
  - the last message $u(l)$ received from $l$ is $\langle 1 \rangle$.

Assume that there is a leaf $l$ such that $P_{u(l)}$ is assigned to a value in $\{l, \perp\}$ each time $u(l)$ executes an $R$ event during the time interval $[t', t' + 2\mathcal{D} \times (X + 2I_{\max} + 2)]$. Then, at time $t' + 2\mathcal{D} \times (X + 2I_{\max} + 2) = (4\mathcal{D}+1) \times (X + 2I_{\max} + 2) + I_{\max} + 1$, which is less than $3\mathcal{D}^2 \times (X + 2I_{\max} + 2)$ time units, the system is in a terminal configuration, by Lemma 5. Otherwise, from time $t'' = (4\mathcal{D}+1) \times (X + 2I_{\max} + 2) + I_{\max} + 1$, for every leaf $l$, we have $P_{u(l)} \neq l$ forever (because for every leaf $l$, $A_{u(l)}[l] = 1$ forever from time $t'$). From that point all the leaves are neutral in the behavior of their unique neighbor (as in the correctness proof), hence we can apply the induction hypothesis on the subtree induced by $V \setminus Leaves_T$ whose diameter is $\mathcal{D} - 2$: the system reaches a terminal configuration within $(4\mathcal{D} + 1) \times (X + 2I_{\max} + 2) + I_{\max} + 1 + 3 \times (\mathcal{D} - 2)^2 \times (X + 2I_{\max} + 2)$ time units, which is less than $3\mathcal{D}^2 \times (X + 2I_{\max} + 2)$ time units.

$\square$

### 5.3 Message Complexity

By definition of the algorithm, every process $p$ sends one message per neighbor at each $R$ event only, so by hypothesis, at most at each time unit. Hence, at most $\sum_{p \in V} \delta_p = 2 \times |E| = 2n - 2$ messages are sent at each time unit, and from Theorem 3, we can deduce the following corollary:

**Corollary 1** *During the stabilization phase of Algorithm $\mathcal{L}$, at most $(6n-6) \times \mathcal{D}^2 \times (X + 2I_{\max} + 2)$ messages are sent.*

## 6 Conclusion

We have proposed a deterministic silent self-stabilizing algorithm for the weak leader election problem in anonymous trees. Our algorithm is designed in the message passing model, requires only $O(1)$ bits of memory per edge, and does not necessitate the *a priori* knowledge of any global parameter on the network. Our algorithm also tolerates frequent message lost, duplication, and reordering. We have proven a bound on the stabilization time of our solution, namely $O((X + I_{\max})\mathcal{D}^2)$ time units.

In future work, we would like to investigate solutions for the weak leader election problem in other classes of anonymous networks, where (deterministic) leader election is impossible. Of course, the definition will have to be relaxed, in a sense that the goal will be to elect the smallest possible subset of neighbors, instead of at most two neighbors.

## References

[1] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[2] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[3] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):69–89, 1996.

[4] Ajoy Kumar Datta and Lawrence L. Larmore. Leader election and centers and medians in tree networks. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 113–132, 2013.

[5] Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, and Nicola Santoro. Sorting and election in anonymous asynchronous rings. *J. Parallel Distrib. Comput.*, 64(2):254–265, 2004.

[6] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.

[7] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.

[8] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *J. Parallel Distrib. Comput.*, 62(5):961–981, 2002.

[9] Gheorghe Antonoiu and Pradip K. Srimani. A self-stabilizing distributed algorithm to find the center of a tree graph. *Parallel Algorithms Appl.*, 10(3-4):237–248, 1997.

[10] Gheorghe Antonoiu and Pradip K. Srimani. A self-stabilizing distributed algorithm to find the median of a tree graph. *J. Comput. Syst. Sci.*, 58(1):215–221, 1999.

[11] Steven C. Bruell, Sukumar Ghosh, Mehmet Hakan Karaata, and Sriram V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM J. Comput.*, 29(2):600–614, 1999.

[12] Ajoy Kumar Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Constant space self-stabilizing center finding in anonymous tree networks. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 38:1–38:10, 2015.

[13] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

[14] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[15] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[16] Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[17] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In *WSS*, pages 62–69, 1999.

[18] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.

[19] Anish Arora and Mikhail Nesterenko. Unifying stabilization and termination in message-passing systems. *Distributed Computing*, 17(3):279–290, 2005.

[20] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Trans. Dependable Sec. Comput.*, 4(3):180–190, 2007.

[21] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *FOCS*, pages 268–277, 1991.

[22] Gerard Tel. *Introduction to distributed algorithms (2nd Ed.)*. Cambridge University Press, 2000.