Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions

Stéphane Devismes\textsuperscript{1}  Colette Johnen\textsuperscript{2}  David Ilcinkas\textsuperscript{2}

\textsuperscript{1} Univ. Grenoble Alpes, VERIMAG, 38000 Grenoble, France
\textsuperscript{2} CNRS & Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France

Meeting DESCARTES, March 28 2018, Paris
Self-Stabilization
Self-stabilization

[Dijkstra, ACM Com., 74]

Configurations

Legitimate

Illegitimate

Transient faults

Time

Legitimate configurations

Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
Self-stabilization

[Dijkstra, ACM Com., 74]

![Diagram showing self-stabilization process]

- Legitimate configurations
- Illegitimate configurations
- Transient faults

Time

Legitimate configurations
Self-stabilization

[Dijkstra, ACM Com., 74]
Self-stabilization

[Dijkstra, ACM Com., 74]

![Diagram showing self-stabilization with transient faults, legitimate, and illegitimate configurations over time.](image)

- Legitimate configurations:
  - Stabilization time
  - Transient faults

- Illegitimate configurations:
  - Stabilization time

Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
A silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the registers used by the algorithm remain fixed.
A silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the registers used by the algorithm remain fixed.

Advantages:

- Silence implies more simplicity in the algorithm design (classically used in compositions).
- A silent algorithm may utilize less communication operations and communication bandwidth.
- Well-suited to compute distributed data structures such as spanning trees.
Model
Abstraction of the message-passing model
Abstraction of the message-passing model

Locally shared registers (variables) instead of communication links

A process can only read its variables and that of its neighbors.
Locally Shared Memory Model with Composite Atomicity

Configuration
Locally Shared Memory Model with Composite Atomicity

Atomic Step

- Reading of the variables of the neighbors
Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
- Daemon election: models the asynchronism
Locally Shared Memory Model with Composite Atomicity

Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
- Daemon election: models the asynchronism
- Update of the local states
Daemons

- Synchronous
- Central / Distributed
- Fairness: Strongly Fair, Weakly Fair, Unfair

**Distributed unfair daemon**: no constraint, except progress!
Complexity

<table>
<thead>
<tr>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory requirement in bits.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(mainly stabilization time)</td>
</tr>
</tbody>
</table>

**Rounds:** execution time *according to the slowest process.*

Essentially similar to the notion of (asynchronous) rounds in message-passing models.

**Moves:** local state updates.

Rather unusual.
Rounds

Processes

Time

1st round

2nd round

Key: Enabled 🟠 Activated ⚫ Neutralized ⭐

Devismes et al
Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions

10/33
Complexity in moves: “a measure of energy”

The stabilization time in moves
- captures the amount of computations an algorithm needs to recover a correct behavior.
Complexity in moves: “a measure of energy”

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.
Complexity in moves: “a measure of energy”

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.

**Contraposition:** If an algorithm is self-stabilizing, for example, under a weakly fair daemon, but not under an unfair one, then its stabilization time in moves cannot be bounded.
The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.

Contraposition: If an algorithm is self-stabilizing, for example, under a weakly fair daemon, but not under an unfair one, then its stabilization time in moves cannot be bounded.

This means that there are processes whose moves do not make the system progress in the convergence: these processes waste computation power and so energy.
Several *a posteriori* analyses show that (classical) self-stabilizing algorithms that work under a distributed unfair daemon have an exponential stabilization time in moves in the worst case.

- BFS spanning tree construction of Huang and Chen [*Devismes and Johnen, JPDC 2016*]
- Leader election of Datta, Larmore, Vemula [*Durand et al, Inf. & Comp. 2017*]
- . . .
General Schemes for Self-Stabilization
The general transformer of [Katz & Perry, Dist. Comp. 93]: not efficient, the purpose is only to demonstrate the feasibility of the transformation (characterization).
Related Work

- The general transformer of [Katz & Perry, Dist. Comp. 93]: not efficient, the purpose is only to demonstrate the feasibility of the transformation (characterization).

- Proof labeling scheme [Korman et al, Dist. Comp. 2010]: restricted class of self-stabilizing algorithms (silent algorithms), stabilization time linear in $n$. No move complexity analysis.

- [Devismes et al, TAAS 2009]: restricted class of self-stabilizing algorithms (wave algorithms), stabilization time linear in $n$ and polynomial in moves.
Related Work

- The general transformer of [Katz & Perry, Dist. Comp. 93]: not efficient, the purpose is only to demonstrate the feasibility of the transformation (characterization).

- Proof labeling scheme [Korman et al, Dist. Comp. 2010]: restricted class of self-stabilizing algorithms (silent algorithms), stabilization time linear in $n$. No move complexity analysis.

- [Devismes et al, TAAS 2009]: restricted class of self-stabilizing algorithms (wave algorithms), stabilization time linear in $n$ and polynomial in moves.

Here, we restrict our study to silent spanning-tree-like data structure (i.e., trees or forests).
Our contribution
Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

Theorem 1

Scheme is silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*

Theorem 2

The stabilization time in rounds of Scheme is at most $4n_{\text{maxCC}}$, where $n_{\text{maxCC}}$ is the maximum number of processes in a connected component.

Theorem 3

When all weights are strictly positive integers bounded by $W_{\text{max}}$, the stabilization time of Scheme in moves is at most $(W_{\text{max}}(n_{\text{maxCC}} - 1)^2 + 5)(n_{\text{maxCC}} + 1)n$.

* n.b., the topologies are not necessarily connected. Disconnection may be due to a transient fault.
Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

**Theorem 1**

Scheme is *silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*

**Theorem 2**

*The stabilization time in rounds of Scheme is at most $4n_{\text{maxCC}}$, where $n_{\text{maxCC}}$ is the maximum number of processes in a connected component.*

---

* *n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.
Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

**Theorem 1**

Scheme is *silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*

**Theorem 2**

*The stabilization time in rounds of Scheme is at most* $4n_{\text{maxCC}}$, *where* $n_{\text{maxCC}}$ *is the maximum number of processes in a connected component.*

**Theorem 3**

*When all weights are strictly positive integers bounded by* $W_{\text{max}}$, *the stabilization time of Scheme in moves is at most* $(W_{\text{max}}(n_{\text{maxCC}} - 1)^2 + 5)(n_{\text{maxCC}} + 1)n$.

*n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.
Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (+ a spanning tree rooted at each leader): $O(n_{\text{maxCC}}^2 n)$ moves
  $\approx$ the best known move complexity [Durand et al, Inf & Comp 2017]

† Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.
‡ With explicit parent pointers.
Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (plus a spanning tree rooted at each leader): $O(n_{\text{maxCC}}^2 n)$ moves
  $\approx$ the best known move complexity [Durand et al, Inf & Comp 2017]

- Given an input, spanning forest with non-rooted components detection\(^\dagger\): $O(n_{\text{maxCC}} n)$ moves
  $\approx$ the best known move complexity for spanning tree construction [Cournier, SIROCCO 2009] \(^\ddagger\)

\(^\dagger\)Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.

\(^\ddagger\)With explicit parent pointers.
Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (+ a spanning tree rooted at each leader): $O(n_{\text{maxCC}}^2 n)$ moves ≈ the best known move complexity [Durand et al, Inf & Comp 2017]

- Given an input, spanning forest with non-rooted components detection†: $O(n_{\text{maxCC}} n)$ moves ≈ the best known move complexity for spanning tree construction [Cournier, SIROCCO 2009] ‡

- In assuming a rooted network, shortest-path spanning tree with non-rooted components detection: $O(W_{\text{max}} n_{\text{maxCC}}^3 n)$ moves ($W_{\text{max}}$ is the maximum weight of an edge) ≈ the best known move complexity [Devismes et al, OPODIS 2016]

† Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.
‡ With explicit parent pointers.
The problem
Inputs (constants)

\[ \textit{canBeRoot}_u : \text{ true if } u \text{ is candidate to be root.} \]
Inputs (constants)

$canBeRoot_u$: true if $u$ is candidate to be root.

In a terminal configuration, every tree root satisfies $canBeRoot$, but the converse is not necessarily true.
Inputs (constants)

\textit{canBeRoot}_u: \text{ true if } u \text{ is candidate to be root.}

For every connected component \( GC \), if there is at least one candidate \( u \in GC \), then at least one process of \( GC \) should be a tree root in a terminal configuration.
Inputs (constants)

$canBeRoot_u$: true if $u$ is candidate to be root.

If there is no candidate in a connected component, all processes of the component should converge to a particular terminal state notifying that it detects the absence of candidate.

(non-rooted components detection)
Inputs (constants)

$canBeRoot_u$: true if $u$ is candidate to be root.

$uname_u$: the name of $u$. 
**Inputs (constants)**

\( canBeRoot_u \): true if \( u \) is candidate to be root.

\( pname_u \): the name of \( u \).

\( pname_u \in IDs \), where \( IDs = \mathbb{N} \cup \{\bot\} \) is totally ordered by \( < \) and \( \min_<(IDs) = \bot \).
Inputs (constants)

$canBeRoot_u$: true if $u$ is candidate to be root.

$pname_u$: the name of $u$.

Two considered cases:
- $\forall v \in V, pname_v = \bot$.
- $\forall u, v \in V, pname_u \neq \bot \land (u \neq v \Rightarrow pname_u \neq pname_v)$
Weights

- $\omega_u(v) \in \text{DistSet}$ denotes the weight of the arc $(u, v)$
- $(\text{DistSet}, \oplus, \prec)$ is an ordered magma:
  - $\oplus$ is a closed binary operation on $\text{DistSet}$
  - $\prec$ is a total order on this set
  - $\forall (u, v), \forall d \in \text{DistSet}, d \prec d \oplus \omega_u(v)$
- $\text{distRoot}(u)$: the distance value of $u$ is $u$ is a root
- $\text{P\_nodeImp}(u)$ is a local predicate which is true is $u$ should move to improve the solution
Variables

\( st_u \in \{ I, C, EB, EF \} \)

\( parent_u \in \{ \perp \} \cup Lbl \): parent in the tree

\( d_u \in DistSet \): distance to the root
Variables

\( st_u \in \{I, C, EB, EF\} \)

Normal behavior

\( I : \text{Isolated} \)
\( C : \text{Correct} \ (\text{belong to a tree}) \)

\( parent_u \in \{\bot\} \cup Lbl: \) parent in the tree

\( d_u \in DistSet: \) distance to the root
Variables

\[ st_u \in \{ I, C, EB, EF \} \]

In a terminal configuration, if \( V_u \) contains a candidate, then \( st_u = C \), otherwise \( st_u = I \).

\[ parent_u \in \{ \perp \} \cup \text{Lbl}: \text{ parent in the tree} \]

\[ d_u \in \text{DistSet}: \text{ distance to the root} \]
Variables

\[ st_u \in \{I, C, EB, EF\} \]

Correction mechanism

- \textit{EB}: Error Broadcast
- \textit{EF}: Error Feedback

\[ parent_u \in \{\bot\} \cup Lbl: \text{ parent in the tree} \]

\[ d_u \in DistSet: \text{ distance to the root} \]
Instances
Leader Election

Inputs:

- $canBeRoot_u$ is true for any process,
- $pname_u$ is the identifier of $u$ (n.b., $pname_u \in \mathbb{N}$)
- $\omega_u(v) = (\bot, 1)$ for every $v \in \Gamma(u)$

Ordered Magma:

- $DistSet = IDs \times \mathbb{N}$
  for every $d = (a, b) \in DistSet$, we let $d.id = a$ and $d.h = b$.
- $(id1, i1) \oplus (id2, i2) = (id1, i1 + i2)$;
- $(id1, i1) \prec (id2, i2) \equiv
  (id1 < id2) \lor [(id1 = id2) \land (i1 < i2)]$
- $distRoot(u) = (pname_u, 0)$

Predicate:

- $P_{nodeImp}(u) \equiv ((\exists v \in \Gamma(u) \mid st_v = C \land d_v.id < d_u.id)) \lor (canBeRoot_u \land distRoot(u) \prec d_u)$
Shortest-Path Spanning Tree

Inputs:

- $canBeRoot_u$ is false for any process except for $u = r$,
- $pname_u$ is $\perp$, and
- $\omega_u(v) = \omega_v(u) \in \mathbb{N}^*$, for every $v \in \Gamma(u)$.

Ordered Magma:

- $DistSet = \mathbb{N}$,
- $i_1 \oplus i_2 = i_1 + i_2$,
- $i_1 \prec i_2 \equiv i_1 < i_2$, and
- $distRoot(u) = 0$.

Predicate:

- $P_{nodeImp}(u) \equiv$
  
  $(\exists v \in \Gamma(u) \mid st_v = C \land d_v \oplus \omega_v(v) \prec d_u)$
  
  $\lor$
  
  $canBeRoot_u \land distRoot(u) \prec d_u$
Our solution in a nutshell
Typical Execution
Any candidate $u$ executes $R_R$: $st_u \leftarrow C$, $d_u \leftarrow \text{distRoot}(u)$, $\text{parent}_u \leftarrow \bot$
Typical Execution

Any non-candidate $v$ executes $R_R$ when it finds a neighbor with status $C$: $st_u \leftarrow C$, select a parent, and $d_v \oplus \omega_u(v)$ if it chooses $v$ as a parent.
In parallel, rules $R_u$ are executed to reduce the weight of the trees: when a process $u$ with status $C$ satisfies $P_{\text{nodelmp}}(u)$, this means that $u$ can reduce $d_u$ by selecting another neighbor with status $C$ as parent.
A candidate can lose its tree root condition using $R_R$, if it finds a sufficiently good parent in its neighborhood.
Typical Execution

Overall, within at most \( n_{\text{max}} \) rounds, a terminal configuration is reached.
Abnormal Roots

Inconsistencies are detected by some processes called Abnormal Roots

A process $u$ is an abnormal root if $u$ is not a normal root, $st_u \neq I$, and satisfies one of the following four conditions:

- its parent pointer does not designate a neighbor,
- its distance $d_u$ is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

An abnormal root $u$ is alive if $st_u \neq EF$

An abnormal tree is a tree root at an abnormal root.

An abnormal tree is alive if it contains a node $v$ such that $st_v \neq EF$

Main result:
No abnormal alive root (resp. tree) is created during the execution.

$^\S$A normal root is any process $v$ such that $canBeRoot_v \land st_v = C \land parent_v = \perp \land d_v = distRoot(v)$. 

Devismes et al
Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
Abnormal Roots

Inconsistencies are detected by some processes called **Abnormal Roots**

A process $u$ is an **abnormal root** if $u$ is not a normal root$^\S$, $st_u \neq I$, and satisfies one of the following four conditions:

- its parent pointer does not designate a neighbor,
- its distance $d_u$ is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

An abnormal root $u$ is **alive** if $st_u \neq EF$

An **abnormal tree** is a tree root at an abnormal root.

An abnormal tree is **alive** if it contains a node $v$ such that $st_v \neq EF$

---

$^\S$A normal root is any process $v$ such that $\text{canBeRoot}_v \land st_v = C \land parent_v = \bot \land d_v = \text{distRoot}(v)$. 
Abnormal Roots

Inconsistencies are detected by some processes called Abnormal Roots

A process $u$ is an abnormal root if $u$ is not a normal root, $st_u \neq I$, and satisfies one of the following four conditions:

- its parent pointer does not designate a neighbor,
- its distance $d_u$ is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

An abnormal root $u$ is alive if $st_u \neq EF$

An abnormal tree is a tree root at an abnormal root.

An abnormal tree is alive if it contains a node $v$ such that $st_v \neq EF$

Main result: No abnormal alive root (resp. tree) is created during the execution.

---

$^\S$A normal root is any process $v$ such that $canBeRoot_v \land st_v = C \land parent_v = \bot \land d_v = distRoot(v)$. 

Devismes et al. Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
Abnormal trees removal

Freeze before Remove

Variable $st_u \in \{I, C, EB, EF\}$

- **I** means *Isolated*
  - a process of status $I$ can join a tree
- **C** means *correct*
  - only processes of status $C$ in a tree can modify their parent pointers and
  - only by choosing a neighbor of status $C$ as parent
- **EB**: Error Broadcast
- **EF**: Error Feedback
Freeze before remove

The definition of abnormal root should take into account possible inconsistencies of variables.

Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
The definition of abnormal root should take to possible inconsistencies of variables into account!
The definition of abnormal root should take into account possible inconsistencies of variables.
The definition of abnormal root should take into account possible inconsistencies of variables!
Freeze before remove

The definition of abnormal root should take into account possible inconsistencies of variables.

Devismes et al. Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
Freeze before remove

The definition of abnormal root should take into account possible inconsistencies.
Freeze before remove

The definition of abnormal root should take into account possible inconsistencies of variables.
The definition of abnormal root should take to possible inconsistencies of variables $st$ into account!
Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\text{maxCC}}$
Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\text{maxCC}}$
- **Cleaning:**
  - EB-wave: $n_{\text{maxCC}}$
  - EF-wave: $n_{\text{maxCC}}$
  - R-wave: $n_{\text{maxCC}}$
Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\text{maxCC}}$
- **Cleaning:**
  - EB-wave: $n_{\text{maxCC}}$
  - EF-wave: $n_{\text{maxCC}}$
  - R-wave: $n_{\text{maxCC}}$

- **Building of the Spanning Tree:** $n_{\text{maxCC}}$ (like in the typical execution)
Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\text{maxCC}}$
- **Cleaning:**
  - EB-wave: $n_{\text{maxCC}}$
  - EF-wave: $n_{\text{maxCC}}$
  - R-wave: $n_{\text{maxCC}}$
- **Building of the Spanning Tree:** $n_{\text{maxCC}}$ (like in the typical execution)

$$O(4n_{\text{maxCC}}) \text{ rounds}$$
Let $GC$ be a connected component of $G$.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration $\gamma$, $u$ is an alive abnormal root, or $canBeRoot_u \land distRoot(u) \prec d_u \land st_u = C$ holds.

**Second case:** $u$ is candidate and can improve by becoming a root ($RU$)
Stabilization Time in Moves (1/2)

Let $GC$ be a connected component of $G$.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration $\gamma$, $u$ is an alive abnormal root, or $canBeRoot_u \land distRoot(u) \prec d_u \land st_u = C$ holds.

**Second case:** $u$ is candidate and can improve by becoming a root ($R_U$)

If a process satisfies one of these two conditions, then it does so from the beginning of the execution.

Let $e = \gamma_0, \cdots, \gamma_i$ be an execution: $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$. 
Let $GC$ be a connected component of $G$.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration $\gamma$, $u$ is an alive abnormal root, or $canBeRoot_u \land distRoot(u) \prec d_u \land st_u = C$ holds.

**Second case:** $u$ is candidate and can improve by becoming a root ($RU$)

If a process satisfies one of these two conditions, then it does so from the beginning of the execution.

Let $e = \gamma_0, \cdots, \gamma_i$ be an execution: $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$.

The size of $SL$ decreases

$\rightarrow$ At most $n_{\text{maxCC}} + 1$ GC-segments in $GC$
Let $u$ be any process of GC. We proved that the sequence of rules executed by $u$ during a GC-segment belongs to the following language:

$$(R_I + \varepsilon)(R_R + \varepsilon)(R_U)^*(R_{EB} + \varepsilon)(R_{EF} + \varepsilon).$$

**Theorem 4**

If the number of $R_U$ executions during a GC-segment by any process of GC is bounded by $\text{nb\_UN}$, then the total number of moves in any execution is bounded by $(\text{nb\_UN} + 4)(n_{\text{maxCC}} + 1)n$. 
Let $u$ be any process of GC. We proved that the sequence of rules executed by $u$ during a GC-segment belongs to the following language:

$$(R_I + \varepsilon)(R_R + \varepsilon)(R_U)^*(R_{EB} + \varepsilon)(R_{EF} + \varepsilon).$$

**Theorem 4**

*If the number of $R_U$ executions during a GC-segment by any process of GC is bounded by $nb\_UN$, then the total number of moves in any execution is bounded by $(nb\_UN + 4)(n_{maxCC} + 1)n$. *

$nb\_UN$ is necessarily defined because $d_u$ decreases at each $R_U(u)$ in a GC-segment.

**Theorem 5**

*When all weights are strictly positive integers bounded by $W_{max}$, $nb\_UN \leq W_{max}(n_{maxCC} - 1)^2 + 1$*

Technical report available online:
https://hal.archives-ouvertes.fr/hal-01667863.