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

ICDCN, January 6th 2019, Bangalore (India)
Self-Stabilization
Self-stabilization

[Devismes et al.
Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions]
Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
Self-stabilization

[Dijkstra, ACM Com., 74]
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

- Reading of the variables of the neighbors
- Enabled nodes
- Daemon selection: models the asynchronism
- Update of the local states
Atomic Step

- Reading of the variables of the neighbors
Locally Shared Memory Model with Composite Atomicity

Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
Locally Shared Memory Model with Composite Atomicity

Atomic Step

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

Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
- Daemon selection: 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

Space

Memory requirement in bits.

Time

(mainly stabilization time)

**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

Key:
- Enabled
- Activated
- Neutralized

Devismes et al. Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions
The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
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.**
Complexity in moves: unusual

Several \textit{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 \cite{Devismes and Johnen, JPDC 2016}
- Leader election of Datta, Larmore, Vemula \cite{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).
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.*

* 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* $4n_{maxCC}$, *where* $n_{maxCC}$ *is the maximum number of processes in a connected component.*

**Theorem 3**

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

---

*n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.
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]

---

† 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 (+ 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
  $\approx$ 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
  $\approx$ the best known move complexity for spanning tree construction [Courrier, 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)
  $\approx$ 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)

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

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

In a terminal configuration, every tree root satisfies \( \text{canBeRoot} \), but the converse is not necessarily true.
\( \text{canBeRoot}_u \): true if \( u \) 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)

\(\text{canBeRoot}_u\): 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.

\( p\text{name}_u \): 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

- \( \omega_u(v) \in DistSet \) denotes the weight of the arc \((u, v)\)

- \((DistSet, \oplus, \prec)\) is an ordered magma:
  - \(\oplus\) is a closed binary operation on \(DistSet\)
  - \(\prec\) is a total order on this set
  - \(\forall (u, v), \forall d \in DistSet, d \prec d \oplus \omega_u(v)\)

- \(distRoot(u)\): the distance value of \(u\) is \(u\) is a root

- \(P_{nodeImp}(u)\) is a local predicate which is true is \(u\) should move to improve the solution
Variables

\[ \text{Variables:} \]

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

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

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

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

Normal behavior:

- \( I : Isolated \)
- \( C : Correct \) (belong to a tree)

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

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

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

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

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

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

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

Correction mechanism

- **EB**: Error Broadcast
- **EF**: Error Feedback

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

\[ d_u \in DistSet: \text{ distance to the root} \]
Instances
Shortest-Path Spanning Tree

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

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

Predicate:
- $P_{nodeImp}(u) \equiv$
  \begin{align*}
  (\exists v \in \Gamma(u) \mid st_v = C \land d_v \oplus \omega_u(v) < d_u) \\
  \lor \\
  canBeRoot_u \land distRoot(u) < d_u
  \end{align*}
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 $d.h = b$.
- $(id_1, i_1) \oplus (id_2, i_2) = (id_1, i_1 + i_2)$;
- $(id_1, i_1) \prec (id_2, i_2) \equiv$
  $(id_1 < id_2) \lor [(id_1 = id_2) \land (i_1 < i_2)]$
- $distRoot(u) = (pname_u, 0)$

Predicate:

- $P\_nodeImp(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \land d_{v.id} < d_{u.id}) \lor distRoot(u) \prec d_u$
Our solution in a nutshell
Typical Execution
Typical Execution

Any candidate $u$ executes $R_R$: either $u$ becomes a root ($st_u \leftarrow C$, $d_u \leftarrow distRoot(u)$, $parent_u \leftarrow \bot$) or join an existing tree rooted to another candidate, if it is “better” ($u$ selects a neighbor $v$ in that tree as parent, and $d_u \leftarrow d_v \oplus \omega_u(v)$)
Any non-candidate $v$ executes $R_R$ when it finds a neighbor with status $C$: $st_u \leftarrow C$, select a parent $v$ among its neighbors of status $C$, and $d_u \leftarrow d_v \oplus \omega_u(v)$.
In parallel, rules $R_u$ are executed to reduce the weight of the trees: when a process $u$ with status $C$ satisfies $P_{nodeImp}(u)$, 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 $R_U$, if it finds a sufficiently good parent in its neighborhood.
Overall, within at most $n_{\text{maxCC}}$ 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 three 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.

---

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

---

§A normal root is any process $v$ such that $canBeRoot_v \land st_v = C \land parent_v = \bot \land d_v = distRoot(v)$. 
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 only by choosing a neighbor of status $C$ as parent, or
    - becoming a root

- $C$ 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
The definition of abnormal root should take 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.
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
The definition of abnormal root should take possible inconsistencies of variables into account!

Devismes et al. Silently Self-Stabilizing Scheme for Spanning-Tree-like Constructions
The definition of abnormal root should take into account possible inconsistencies.
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/forest: $n_{\text{maxCC}}$ (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/forest: $n_{\text{maxCC}}$ (like in the typical execution)

$O(4n_{\text{maxCC}})$ 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 $\text{canBeRoot}_u \land \text{distRoot}(u) \prec d_u \land st_u = C$ holds.

**Second case:** $u$ is candidate and can improve by becoming a root ($R_U$)
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: $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$. 
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 ($RU$)

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

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

→ At most $n_{maxCC} + 1$ 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 $nb\_UN$, then the total number of moves in any execution is bounded by $(nb\_UN + 4)(n_{maxCC} + 1)n$. 

**Theorem 5**

When all weights are strictly positive integers bounded by $W_{max}$, 

$nb\_UN \leq W_{max}(n_{maxCC} - 1) + 1$.
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:

\[
(R_I + \varepsilon)(R_R + \varepsilon)(R_U)^\star(R_{EB} + \varepsilon)(R_{EF} + \varepsilon).
\]

**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 \( W_{max} \), \( nb\_UN \leq W_{max}(n_{maxCC} - 1)^2 + 1 \).*
Conclusion
Conclusion

Our scheme is versatile since it can be applied to build **efficient silent self-stabilizing algorithms** for

- leader election (+ spanning tree of type arbitrary, BFS, DFS, ...), or
- spanning tree or forest constructions of type:
  - arbitrary
  - BFS
  - DFS
  - Shortest-Path
  - ...

in

- identified or
- semi-anonymous (e.g., rooted)

networks.
Thank you for your attention