SELF-STABILIZING NETWORK ORIENTATION
ALGORITHMS IN ARBITRARY ROOTED NETWORKS

AJOY K. DATTA, SHIVASHANKAR GURUMURTHY
FRANCK PETIT, VINCENT VILLAIN

Abstract. We propose the first self-stabilizing protocols for orienting arbitrary networks. We present three deterministic distributed network orientation protocols in arbitrary and asynchronous rooted networks. All the protocols set up a chordal sense of direction in the network. The protocols are self-stabilizing, meaning that starting from an arbitrary state, the protocols are guaranteed to reach a state, in which all node labels and edge labels are valid (meaning, they satisfy the specification of the orientation problem). We first propose a very simple orientation protocol, called PEL. Algorithm PEL assumes that every processor knows the size \( n \) of the network and the processors are already assigned valid node labels. This protocol stabilizes in only 1 round. For the two other protocols, we assume that the network is rooted. We then compose these protocols with Algorithm PEL to obtain the corresponding orientation protocols for any arbitrary network. The second protocol is based on a sequential node naming and size computation algorithm which assumes an underlying depth-first token circulation protocol. When this protocol is composed with Algorithm PEL, we obtain an orientation algorithm which stabilizes in \( O(n \times d) \) rounds where \( d \) is the diameter of the network. This protocol involves designing a parallel node naming and size computation algorithm, and assumes the existence of an underlying spanning tree protocol. The third protocol can be composed with Algorithm PEL to design an orientation algorithm which stabilizes in \( O(d) \) rounds. Although the third protocol assumes the construction of a spanning tree of the rooted network, it orients all edges—both tree and non-tree—of the network.

Keywords: Network orientation, self-stabilization, sense of direction, spanning tree, token circulation.

1. Introduction

Modern distributed systems have the inherent problem of faults. The quality of a distributed system design depends on its tolerance to faults that may occur at various components of the system. Many fault tolerant schemes have been proposed and implemented, but the most general technique to design a

Ajoy K. Datta was supported in part by a sabbatical leave grant from University of Nevada, Las Vegas. Franck Petit and Vincent Villain were supported in part by the Pôle de Modélisation de Picardie, France.

Studia Informatica Universalis
system that tolerates arbitrary transient faults is self-stabilization [6]. A self-stabilizing protocol guarantees that, starting from an arbitrary initial state, the system converges to a desirable state in finite time.

The network orientation problem concerns the assignment of different labels or directions to the edges adjacent to processors in a globally consistent manner. The label of an edge indicates which direction in the network the edge leads to. The labels can be used in many applications, such as routing and traversal in networks.

It was demonstrated by Santoro [16] that the availability of an orientation in a network decreases the message complexity of some computations on various topologies. Many subsequent papers have assumed oriented networks in order to reduce the algorithm complexity. Surprisingly, there are very few papers that have addressed the question of how orientations can be computed in networks. Korfhage and Gafni [15] have presented an algorithm to orient directed tori. The orientation problem for tori was also studied by Syrotiuk et al. [17]. There has also been considerable interest in the problem of orienting a ring network [14, 18, 4]. A major work in this area has been reported in [19], which reported studies on orientation of cliques, hypercubes, and tori in both anonymous and non-anonymous networks.

There exist only two articles [3, 14] on the self-stabilizing orientation. The algorithm proposed in [14] is to orient rings, and the one in [3] is a torus orientation algorithm. Both are probabilistic algorithms.

Another related area of research is the Sense Of Direction (SoD) [10], which allows processors to communicate efficiently, by exploiting the topological properties of the network algorithmically. Flocchini, et. al. [10] gave a formal definition of SoD, and also showed a relationship among three factors: the labeling, the topological structure, and the local view that an entity has of the system. Tel [20] has shown how the election problem on rings, hypercubes, and cliques can be solved more efficiently by exploiting the SoD. Also, SoD allows to obtain a logical ring using fewer links than the Eulerian tour of a spanning tree.

In this paper, we propose the first self-stabilizing protocols for orienting an arbitrary rooted network.

The first algorithm (PEL) assumes pre-assigned node labels compatible with the chordal sense of direction and a knowledge of the size of the network. This algorithm allows processors to assign globally consistent edge labels in parallel. This simple protocol stabilizes in only 1 round and requires \(O(\Delta_p \times \log n)\) bits per processor \(p\), where \(\Delta_p\) is the degree of \(p\).
Next we assume only rooted networks and present two methods – both based on computing the name and size of the network. The first scheme, called \( \text{SNSC} \), uses a token to implement a sequential naming and size computation protocol. We assume that an underlying depth-first token circulation protocol is maintained on an arbitrary network. Huang and Chen [13] have presented a non-deterministic depth-first token circulation protocol for a connected network. In a more recent work [5], Datta, et al., proposed a self-stabilizing depth-first token passing protocol. This protocol has a better space complexity than the previous algorithms. In this paper, we use the protocol of [5] as an underlying protocol to maintain the token circulation. Once stabilized, this protocol makes the token follow the same path forever. This is very useful to maintain the same name on the nodes forever (as long as the topology does not change). This protocol does not require an underlying tree to be maintained. Using Algorithm \( \text{PEL} \) in conjunction with this naming and size computation protocol, we obtain an algorithm which stabilizes in \( O(n \times d) \) rounds where \( d \) is the diameter of the network. Algorithm \( \text{SNSC} \) requires only \( O(\log \Delta_p + \log n) \) bits per processor.

The second protocol is based on a composition of three algorithms: a parallel weight computation (\( \text{PWC} \)), a parallel naming (\( \text{PN} \)), and a parallel size broadcast (\( \text{PSB} \)). We obtain an orientation protocol, called \( \text{NOST} \), by composing \( \text{PEL} \), \( \text{PN} \), \( \text{PSB} \), \( \text{PWC} \), and a self-stabilizing spanning tree construction. Using a self-stabilizing spanning tree construction algorithm, (e.g., [1, 2, 9, 12]) this network orientation protocol stabilizes in \( O(d) \) rounds. Algorithms \( \text{PN} \), \( \text{PSB} \), \( \text{PWC} \), and self-stabilizing spanning tree construction algorithm require an extra space of \( O((\Delta_{T_{\text{tree}}} \times \log n) + \Delta_p) \) bits per processor, where \( \Delta_{T_{\text{tree}}} \) is the number of neighbors of \( p \) in the underlying spanning tree. Since all the constituent protocols are silent [7], Algorithm \( \text{NOST} \) is also silent.

In Section 2, we introduce the notion of self-stabilization and network orientation, and give a brief description of the related work in those areas. We also present in Section 2 the problem of network orientation and the model of the distributed system considered in this paper. In Section 3, we present the self-stabilizing \( \text{PEL} \) along with the proof of correctness. The other two orientation protocols along with their correctness proofs are presented in the next two subsequent sections—Sections 4 and 5. Finally, we make some concluding remarks in Section 6.
2. Preliminaries

In this section, we define the distributed systems and programs considered in this paper, and state what it means for a protocol to be self-stabilizing. We also provide definitions for protocol composition which we use for simplifying the design and proofs for our algorithms. We then define the problem of network orientation in arbitrary rooted networks.

2.1. Self-Stabilizing System

A distributed system is an undirected connected graph, \( S = (V, E) \), where \( V \) is a set of processors (\( |V| = n \)) and \( E \) is the set of bidirectional communication links. We consider networks which are asynchronous and rooted, i.e., all processors, except the root are anonymous. We denote the root processor by \( r \). A communication link \((p, q)\) exists iff \( p \) and \( q \) are neighbors. We denote the set of incident edges on a processor \( p \) as \( E_p \), and the edge connecting processor \( p \) with \( q \) as \( E_{p,q} \). Each processor \( p \) maintains its set of neighbors, denoted as \( N_p \). We assume that \( N_p \) is a constant and is maintained by an underlying protocol. The degree of \( p \) is denoted by \( \Delta_p \) and is equal to \( |N_p| \).

The program consists of a set of shared variables (henceforth referred to as variables) and a finite set of actions. A processor can only write to its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of \( p \) can be accessed by \( p \) and its neighbors.

Each action is uniquely identified by a label and is of the following form:

\[
<\text{label}>:: <\text{guard}> \rightarrow <\text{statement}>
\]

The guard of an action in the program of a processor \( p \) is a boolean expression involving the variables of \( p \) and its neighbors. When the guard of an action labeled \( A \) in the program of \( p \) is true, then the processor \( p \) and the action \( A \) is said to be enabled. An action can be executed only if it is enabled. The statement of an action of \( p \) updates some of the variables of \( p \). We assume that the actions are atomically executed: The evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The state of a processor is defined by the values of its variables. The state of a system is a product of the states of all processors (\( \in V \)). In the following, we refer to the state of a processor and system as a \((\text{local})\) state and configuration, respectively. Let a distributed protocol \( P \) be a collection of binary transition relations denoted by \( \rightarrow \), on \( \mathcal{C} \), the set of all possible configurations.
of the system. A computation of a protocol $\mathcal{P}$ is a maximal sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$ (a single computation step) if $\gamma_{i+1}$ exists, or $\gamma_i$ is a terminal configuration. Maximal means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. When a computation is finite, we say that the algorithm is silent. The set of computations of a protocol $\mathcal{P}$ in system $S$ starting from a particular configuration $\alpha \in \mathcal{C}$ is denoted by $\mathcal{E}_\alpha$. The set of all possible computations of $\mathcal{P}$ in system $S$ is denoted as $\mathcal{E}$.

We assume a weakly fair and distributed daemon. The weak fairness means that if a processor $p$ is continuously enabled, then $p$ will be eventually chosen by the daemon to execute an action. The distributed daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses a nonempty subset of processors to execute an action.

In order to compute the time complexity measure, we use the definition of round. Given a computation $e$ ($e \in \mathcal{E}$), the first round of $e$ (let us call it $e'$) is the minimal prefix of $e$ containing one (local) atomic step of every continuously enabled processor from the first configuration. Let $e''$ be the suffix of $e$, i.e., $e = e'e''$. Then second round of $e$ is the first round of $e''$, and so on.

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate $P$ defined on the set $\mathcal{X}$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We define a special predicate true as follows: for any $x \in \mathcal{X}$, $x \vdash$ true.

We use the following term, attractor in the definition of self-stabilization.

**Definition 1 (Attractor).** Let $X$ and $Y$ be two predicates of a protocol $\mathcal{P}$ defined on $\mathcal{C}$ of system $S$. $Y$ is an attractor for $X$ if and only if the following condition is true:

$$\forall \alpha \vdash X : \forall e \in \mathcal{E}_\alpha : e = (\gamma_0, \gamma_1, ...) :: \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y.$$ We denote this relation as $X \triangleright Y$.

**Definition 2 (Self-stabilization).** The protocol $\mathcal{P}$ is self-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if there exists a predicate $\mathcal{L}_\mathcal{P}$ (called the legitimacy predicate) defined on $\mathcal{C}$ such that the following conditions hold:

1. $\forall \alpha \vdash \mathcal{L}_\mathcal{P} : \forall e \in \mathcal{E}_\alpha :: e \vdash \mathcal{SP}_\mathcal{P}$ (correctness).
2. $\text{true} \triangleright \mathcal{L}_\mathcal{P}$ (closure and convergence).

SIU 2001
2.2. Protocol Composition

We will use the idea of protocol composition [8, 11, 21] in order to simplify the design and proof of algorithms.

Definition 3 (Collateral composition). Let $S_1$ and $S_2$ be programs such that no variables written by $S_2$ appears in $S_1$. The collateral composition of $S_1$ and $S_2$, denoted as $S_2 \circ S_1$, is the program that has all the variables and all the actions of $S_1$ and $S_2$.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be predicate over the variables of $S_1$ and $S_2$, respectively. In the composite algorithm, $\mathcal{L}_1$ will be established by $S_1$, and subsequently, $\mathcal{L}_2$ will be established by $S_2$. We now define a fair composition w.r.t. both programs, and define what it means for a composite algorithm to be self-stabilized.

Definition 4 (Fair execution). An execution $e$ of $S_2 \circ S_1$ is fair w.r.t. $S_i$ ($i \in \{1, 2\}$) if one of these conditions holds:
1. $e$ is finite;
2. $e$ contains infinitely many steps of $S_i$, or contains an infinite suffix in which no step of $S_i$ is enabled.

Definition 5 (Fair Composition). The composition $S_2 \circ S_1$ is fair w.r.t. $S_i$ ($i \in \{1, 2\}$) if every execution of $S_2 \circ S_1$ is fair w.r.t. $S_i$.

Theorem 1. $S_2 \circ S_1$ stabilizes to $\mathcal{L}_2$ if the following four conditions hold:
1. Program $S_1$ stabilizes to $\mathcal{L}_1$.
2. Program $S_2$ stabilizes to $\mathcal{L}_2$ if $\mathcal{L}_1$ holds.
3. Program $S_1$ does not change variables read by $S_2$ once $\mathcal{L}_1$ holds.
4. The composition is fair w.r.t. both $S_1$ and $S_2$.

2.3. Chordal Sense of Direction

All protocols discussed in this paper set up a chordal sense of direction in the un-oriented network. We use the definition of the chordal sense of direction as in [10]. A chordal sense of direction in a connected undirected graph $S = (V, E)$, with $|V| = n$, is defined by fixing a cyclic ordering of the nodes, and labeling each link by the distance in the above cycle.

Let $\psi : V \rightarrow V$ be a successor function defining a cyclic ordering of the nodes of $S$ and let $\psi^k(p) = \psi^{k-1}(\psi(p))$ for $k > 0$. Let $\delta : V \times V \rightarrow \{0, \ldots, n-1\}$ be the corresponding distance function, i.e., $\delta(p, q)$ is the smallest $k$ such that $\psi^k(p) = q$. The labeling $\pi$ is a chordal labeling iff, $\forall (p, q) \in E_p : \pi_p(p, q) = \delta(p, q)$. 

SIU 2001
Note that \( \psi \) is the function defining the cyclic ordering of the nodes, and the different chordal labeling functions arise from different \( \psi \)'s. Further note that, if the link between \( p \) and \( q \) is labeled by \( d \) at node \( p \), it is labeled by \( n - d \) at node \( q \). An example of chordal sense of direction is shown in Figure 2.1.

![Figure 2.1: Chordal sense of direction.](image)

### 2.4. Specification of the Network Orientation Problem

A labeling of a network is an assignment in every node, of different labels to the edges incident to that node. An orientation of a network is a labeling scheme where the labels satisfy an additional global consistency property. In our case (since we are using a chordal sense of direction), the consistency property is as follows: Each node \( p \) is assigned a unique name \( \eta_p \) from the set \( 0, 1, \ldots, n - 1 \), such that the edge connecting node \( p \) to node \( q \) is labeled \((\eta_p - \eta_q) \mod n\) at node \( p \). We consider a computation \( e \) of the network orientation problem, \( \mathcal{NO} \), to satisfy the specification, \( \mathcal{SP}_{\mathcal{NO}} \), of the problem, if the following conditions are true:

\[
(SP1) \text{ Every node in the network has a unique (permanent) name } \eta_p \text{ in the range } 0, \ldots, n - 1.
\]

\[
(SP2) \forall p \in V : \forall l \in E_{p,q} :: \pi_p[l] = (\eta_p - \eta_q) \mod n.
\]

Note that \( SP1 \) (the unique naming of nodes) guarantees that the assigned edge labels satisfying \( SP2 \) are \textit{locally oriented}, i.e., will be unique locally at the node.
3. Parallel Edge Labeling

In this section, we propose a self-stabilizing edge labeling protocol. The algorithm is parallel, meaning that processors compute their edge labels in parallel if the labels are inconsistent. We first give a general description of the algorithm followed by Algorithm \( \mathcal{PEL} \). We also present the correctness proof for \( \mathcal{PEL} \) algorithm.

3.1. Algorithm \( \mathcal{PEL} \)

The variable \( \pi_p \) is an array that stores the edge label for every edge incident on \( p \). The algorithm uses variables \( \eta_p \) and \( \text{Size}_p \) as constants (they can be computed using the naming and size computation techniques as discussed later). The algorithm consists of a single guarded statement, which checks all the edge labels for correctness, using the predicate \( \text{InvEdgelabel}(p) \). The macro \( \text{Edgelabel}_p \) computes an edge label by computing the difference of the connecting node names modulo \( \text{Size}_p \). The algorithm is presented as Algorithm 3.1.

Algorithm 3.1 (\( \mathcal{PEL} \))  Parallel Edge Labeling.

**Uses**

\( \text{Size}_p, \eta_p : \text{integer} \)

**Variables**

\( \pi_p[1..|\mathcal{N}_p|] : \text{array of integer} \)

**Predicates**

\[ \text{InvEdgelabel}(p) \equiv \exists q \in \mathcal{N}_p : \pi_p[E_{p,q}] \neq (\eta_p - \eta_q) \mod \text{Size}_p \]

**Macro**

\( \text{Edgelabel}_p = \{ \text{for all } q \in \mathcal{N}_p \text{ do } \pi_p[E_{p,q}] := (\eta_p - \eta_q) \mod \text{Size}_p \} \)

**Actions**

\( \text{InvEdgelabel}(p) \rightarrow \text{Edgelabel}_p \)

3.2. Correctness of Parallel Edge Labeling Algorithm

We define the predicates \( \mathcal{L}_{\mathcal{NL}}, \mathcal{L}_S, \) and \( \mathcal{L}_{\mathcal{EL}} \) as follows:

1. \( \alpha \vdash \mathcal{L}_{\mathcal{NL}} \equiv \text{Every node in the network has a unique (permanent) name } \eta_p \text{ in the range } 0, \ldots, n-1. \)
2. \( \alpha \vdash \mathcal{L}_S \equiv \forall p \in V : \text{Size}_p = n. \)
3. \( \alpha \vdash \mathcal{L}_{\mathcal{EL}} \equiv \forall p \in V : \forall l \in E_{p,q} : \pi_p[l] = (\eta_p - \eta_q) \mod \text{Size}_p. \)
Since, in this section, we assume that each processor $p$ is assigned a unique name $\eta_p$ and a knowledge of the size of the network, the following is true:

$$\forall \alpha \in \mathcal{E} : \alpha \vdash L_{NL} \land L_S.$$  

**Lemma 2.** If a processor $p$ is enabled, then it will eventually move once and will not move thereafter.

**Proof.** (1) Assume that $InvEdgelabel(p)$ is false. Since $\eta_p$ and $Size_p$ never change, $InvEdgelabel(p)$ remains false forever. So, $p$ will never be enabled and hence will never move.

(2) Now, we assume that $InvEdgelabel(p)$ is true. If $p$ eventually moves, then, by (1), $p$ will never move again thereafter. If $p$ never moves, then $p$ is enabled forever ($InvEdgelabel(p)$ remains true forever because $\eta_p$ and $Size_p$ never change). By fairness, $p$ eventually moves which contradicts the assumption. $\square$

The following theorem follows directly from Lemma 2:

**Theorem 3.** Algorithm $\mathcal{PEL}$ is self-stabilizing.

**Proof.** Obviously, if $\alpha \vdash L_{EL}$, then no processor is enabled and any $e \in \mathcal{E}_\alpha$ is equal to $\alpha$. So, the correctness is verified.

Let $\alpha$ be a configuration such that $\exists p \in V : \exists l \in E_{p,q} : \pi_p[l] \neq (\eta_p - \eta_q) \mod n$. Then $p$ is enabled. Then by Lemma 2, $p$ will move and will never be enabled. So, in any computation, we reach the final configuration $\vdash L_{EL}$, and the closure and convergence are verified. $\square$

The following corollary is obvious from the proof of Theorem 3:

**Corollary 4.** Algorithm $\mathcal{PEL}$ is silent.

**Note 3.1.** The results in this subsection (i.e., Algorithm $\mathcal{PEL}$) also hold for an unfair daemon. (The unfairness means that even if a processor $p$ is continuously enabled, then $p$ may never be chosen by the daemon if $p$ is not the only enabled processor.)

### 3.3. Space and Time Complexity

The time to stabilize Algorithm $\mathcal{PEL}$ is at most 1 round because all processors can move in parallel. Every node maintains one array variable, $\pi_p$, which has $\Delta_p$ elements. Every element of the array is in $1, ..., n - 1$. Thus, the space complexity for Algorithm $\mathcal{PEL}$ is $O(\Delta_p \times \log n)$ bits per processor.
4. Network Orientation using Depth-First Token Passing

In this section, we propose a self-stabilizing network orientation algorithm using depth-first token circulation protocol. We first present the data structure used by the sequential naming and size computation algorithm, followed by Algorithm \textit{SN\textsc{sc}}. We then define a protocol composition scheme, called \textit{conditional composition}, and show that the conditional composite algorithm \textit{SN\textsc{sctc}} (Sequential Naming, Size Computation, and depth-first Token Circulation) is self-stabilizing. Finally we present Algorithm \textit{NOTC} (Network Orientation based on Token Circulation) using the collateral composition of \textit{SN\textsc{sctc}} and \textit{PEL}, and its correctness proof.

4.1. Algorithm \textit{SN\textsc{sc}}

We use the depth-first token circulation algorithm of [5], which we refer to as Algorithm \textit{DFTC}. Algorithm \textit{SN\textsc{sc}} reads two variables of \textit{DFTC}: \(D_p\) and \(A_p\). The current descendant (ancestor) of a processor is maintained in the variable \(D_p\) (\(A_p\)), where \(D_p(A_p) \in \mathcal{N}_p \cup \perp\). Note that the processors do not maintain \(A_p\). They dynamically compute it as follows: \(A_p = q\), where \(D_q = p\). That is, to compute \(A_p\), processor \(p\) reads \(D_q\) for all neighbors \(q\) and checks to see if \(D_q = p\). Every processor \(p\) also maintains the integer variables \(\text{Cnt}_p\), \(\text{Size}_p\), and \(\eta_p\). \(\text{Cnt}_p\) is used to count the number of nodes visited by the token. \(\text{Size}_p\) refers to the number of processors in the network, i.e., the maximum value of \(\text{Cnt}_p\) in the whole network. \(\eta_p\) maintains the label of the node corresponding to processor \(p\).

Algorithm \textit{SN\textsc{sc}} is shown as Algorithm 4.2. The macro \textit{UpdCnt}, \textit{UpdSize}, and \textit{Nodelabel} is used to update \(\text{Cnt}_p\), update the size of the network, and name the node, respectively.

A node is said to hold a token if the following predicate holds:

\[
\text{Token}(p) \equiv \text{Forward}(p) \lor \text{Backtrack}(p),
\]

where \(\text{Forward}(p)\) and \(\text{Backtrack}(p)\) are predicates from Algorithm \textit{DFTC}. \(\text{Forward}(p)\) is enabled at processor \(p\) when it receives a token for the first time from its parent \(A_p\). The predicate \(\text{Backtrack}(p)\) is true every time the token is backtracked to processor \(p\) from its descendant \(D_p\). For a more detailed description of \(\text{Forward}(p)\) and \(\text{Backtrack}(p)\), refer to [5].

The depth-first token circulation protocol guarantees that every node, during a single complete traversal, will have its \(\text{Forward}(p)\) enabled exactly once. When a node has a token for the first time (in a single traversal), it assigns the next lowest available name as its node label, after consulting its parent. The
Algorithm 4.2 (SNSC) Sequential Naming and Size Computation using Depth-First Token Circulation.

Uses
For any processor \( p \) : \( D_p : \text{integer}; \)
\( \text{Forward}(p), \text{Backtrack}(p) : \text{predicates}; \)

For processors \( p \neq r \) : \( A_p : \text{integer} \)

Constants
For the root \( (p = r) \) : \( \eta_p = 0 \)

Variables
For the root \( (p = r) \) : \( \text{Size}_p, \text{Cnt}_p : \text{integer}; \)
For the other processors : \( \text{Size}_p, \eta_p, \text{Cnt}_p : \text{integer} \)

Macro
\[
\text{UpdSize}_p = \begin{cases}
\text{Size}_p := \text{Cnt}_p; \text{Cnt}_p := 0; \text{if} \ (p = r) \\
\text{Size}_p := \text{Size}_{A_p}; \text{Cnt}_p := \text{Cnt}_{A_p} + 1 \text{ otherwise}
\end{cases}
\]
\[
\text{Nodelabel}_p = \eta_p := \text{Cnt}_p \text{ if} \ (p \neq r)
\]
\[
\text{UpdCnt}_p = \text{Cnt}_p := \text{Cnt}_{D_p}
\]

Actions
\( \text{Forward}(p) \rightarrow \text{UpdSize}_p; \text{Nodelabel}_p \)
\( \text{Backtrack}(p) \rightarrow \text{UpdCnt}_p \)

node then passes the token on to the next node (descendant), if any. Otherwise, it backtracks the token to its parent along with the current Cnt value.

4.2. Algorithm SNSCTC

We use a protocol composition technique, called conditional composition to simplify the design and proof for the SNSCTC algorithm. We first define the notion of conditional composition.

Definition 6 (Conditional Composition). Let \( S_1 \) and \( S_2 \) be programs such that variables written by \( S_2 \) are not referred by \( S_1 \). The conditional composition of \( S_1 \) and \( S_2 \), denoted by \( S_2 \circ_{\mathcal{G}} S_1 \), is a program that satisfies the following conditions:

1. It contains all the variables and actions of \( S_1 \) and \( S_2 \).
2. \( \mathcal{G} \) is a set of predicates and is a subset of the guards of \( S_1 \).
3. Every guard of \( S_2 \) has the form \( g \land h \text{ or } \neg g \land h \) where \( g \) is a logical expression using the guards \( \in \mathcal{G} \).
4. Since some actions of \( S_2 \) may also be enabled when an action of \( S_1 \) is enabled, the order of execution is as follows: the action of \( S_2 \) followed by the action of \( S_1 \) (in the same step).

SIU 2001
Algorithm \(SNSCTC\) is shown as Algorithm 4.3.

**Algorithm 4.3 \((SNSCTC)\)** Sequential Naming, Size Computation, and Depth-First Token Circulation.

\[
SNSC \circ ([Forward, Backtrack]) DFTC
\]

We now explain the order of execution of the actions of Algorithm \(SNSCTC\). The first action of \(SNSC\) is guarded by the \(Forward\) predicate. So, when this predicate is true in \(SNSCTC\), the processor first executes \(UpdSize_p\) and \(Nodelabel_p\), then the action in Algorithm \(DFTC\) corresponding to the predicate \(Forward\) (it sends the token to its first descendant, if any, or to its ancestor, otherwise). We use \(LTC\) to denote the legitimacy predicate of Algorithm \(DFTC\) [5].

We cannot use Theorem 1 directly to prove the correctness of Algorithm \(SNSCTC\) since we used the conditional composition here. But, we can make the following observations:

1. Algorithm \(DFTC\) stabilizes to \(LTC\). (Algorithm \(SNSC\) has no impact on the behavior of \(DFTC\).)
2. Algorithm \(SNSC\) stabilizes to \(LN_L \land LS\) if \(LTC\) holds. (Once stabilized, the token follows the same path forever. Thus, the node labels assigned are permanent after the first normal token circulation, and when the token in the next traversal reaches all processors, every processor knows the exact size of the net.)
3. Algorithm \(DFTC\) DOES change variables read by \(SNSC\) after \(LTC\) holds. So, the third point of Theorem 1 is not satisfied. But, once Algorithm \(DFTC\) stabilizes, (as discussed before) the token always takes the same route. Hence, the counter value every processor uses to name itself will not change after \(DFTC\) stabilizes, i.e., the node naming is permanent. Thus, the variables read by \(SNSC\) after \(LTC\) holds, effectively do not change.
4. The composition is fair w.r.t. both \(DFTC\) and \(SNSC\). (Every time \(Forward\) or \(Backtrack\) is true, the action of both \(DFTC\) and \(SNSC\) is executed.)

From the above observations, we can claim the following result:

**Lemma 5.** Algorithm \(SNSCTC\) stabilizes for the specification defined by the legitimacy predicate \(LN_L \land LS\).
4.3. Space and Time Complexity of Algorithm $SNSCTC$

The time to stabilize Algorithm $SNSCTC$ is $O(n \times d)$. More precisely, Algorithm $DFTC$ requires $O(n \times d)$ to stabilize, and once Algorithm $DFTC$ stabilized, Algorithm $SNSC$ needs only two token circulations to stabilize. The algorithm maintains the node names and size of the network along with the descendant and parent pointers. So, it requires $O(\log \Delta_p + \log n)$ bits per processor.

4.4. Algorithm $NOTC$

Algorithm $NOTC$ (shown as Algorithm 4.4) orients any rooted network. It is a collateral composite algorithm.

Algorithm 4.4 ($NOTC$)  Network Orientation Based on Token Circulation.

$PEL \circ SNSCTC$

We can make the make the following observations:

1. Algorithm $SNSCTC$ stabilizes to $\mathcal{L}_{NL} \wedge \mathcal{L}_S$ (see Lemma 5).
2. Algorithm $PEL$ stabilizes to $\mathcal{L}_{EL}$ if $\mathcal{L}_{NL} \wedge \mathcal{L}_S$ holds (see Theorem 3).
3. Algorithm $SNSCTC$ does not change variables read by $PEL$ once $\mathcal{L}_{NL} \wedge \mathcal{L}_S$ holds.
4. The composition is fair w.r.t. both $SNSCTC$ and $PEL$ (every execution contains an infinite suffix in which no steps of $PEL$ are enabled).

From the above observations, we can apply Theorem 1 to prove the correctness of Algorithm $NOTC$.

Theorem 6. Algorithm $NOTC$ is self-stabilizing for $SP_{NO}$.

4.5. Space and Time Complexity of Algorithm $NOTC$

Since the stabilizing time of $PEL$ is 1, the time complexity measure for this protocol composition is the same as that of $SNSCTC$, i.e. $O(n \times d)$. The number of bits required per processor is $O(\Delta_p \times \log n)$ ($O(\log \Delta_p + \log n)$ for $SNSCTC$ and $O(\Delta_p \times \log n)$ for $PEL$).

SIU 2001
5. Network Orientation Using Spanning Tree Protocol

In this section, we propose a self-stabilizing network orientation protocol that uses an underlying spanning tree protocol (let us call it Algorithm $ST$) to orient the network. We assume that Algorithm $ST$ deterministically maintains a spanning tree of the graph. In $ST$, all processors except the root maintain their ancestor (in the tree), denoted by $A_p$, where $A_p \in \mathcal{N}_p$. The set, $\mathcal{D}_p = \{ q \in \mathcal{N}_p : A_q = p \}$ is called the set of descendants of $p$. Note that the processors do not maintain this variable (or set) $\mathcal{D}_p$, but they dynamically compute it. We also assume that the descendants of $p$ are ordered from 1 through $|\mathcal{D}_p|$.

First, we present two simple algorithms: a weight computation algorithm (Algorithm $PWC$) and a size broadcast algorithm (Algorithm $PSB$). We then give the parallel naming algorithm (Algorithm $PN$). Finally, we present Algorithm $NOST$ (Network Orientation using Spanning Tree protocol) which is a collateral composition of $PEL$, $PN$, $PSB$, $PWC$, and $ST$.

5.1. Algorithms $PWC$ and $PSB$

5.1.1. Algorithm $PWC$

Variable $Weight_p$ maintains the size (or the number of nodes) of the subtree rooted at $p$. The predicate $InvWeight$ becomes true if an internal node or the root detects that its $Weight$ is incorrect. In that situation, the node computes $Weight$ by adding the $Weight$ variables of all its descendants and its own. This proceeds bottom-up on the tree until the root computes the $Weight$ value from its subtrees. The leaf processors maintain their $Weight$ as 1. In at most $h$ rounds, every node will be able to set its $Weight$ variable to the right value. The implementation of $PWC$ is given in Algorithm 5.5.

Lemma 7. Assuming an underlying spanning tree, Algorithm $PWC$ is silent and stabilizes in at most $h$ rounds.

5.1.2. Algorithm $PSB$

This algorithm is shown as Algorithm 5.6. Since the information about the size of the network is used by the orientation algorithm, we use a parallel broadcast algorithm to propagate the size to all the nodes in the tree. We assume that the root knows the weight of the tree as a constant. The root then broadcasts this value to all nodes in the tree.
Algorithm 5.5 (PWC) Parallel Weight Computation using Spanning Tree.

**Uses**
- For the root \((p = r)\) and the internal processors: \(D_p: \text{set of integer}\)

**Constants**
- For the leaf processors: \(Weight_p = 1\)

**Variables**
- For the root \((p = r)\) and the internal processors: \(Weight_p: \text{integer}\)

**Predicates**
- \(InvWeight(p) \equiv Weight_p \neq 1 + \sum_{q \in D_p} Weight_q\)

**Actions**
- \(InvWeight(p) \rightarrow Weight_p := 1 + \sum_{q \in D_p} Weight_q\)

Algorithm 5.6 (PSB) Parallel Size Broadcast using Spanning Tree and Weight Computation.

**Uses**
- For the root \((p = r)\): \(Weight_p: \text{integer}\)
- For the leaf and internal processors: \(A_p: \text{integer}\)

**Constants**
- For the root \((p = r)\): \(Size_p = Weight_p\)

**Variables**
- For the leaf and internal processors: \(Size_p: \text{integer}\)

**Actions**
- \(\{\text{For the leaf and the internal processors}\}\)
  - \(Size_p \neq Size_{A_p} \rightarrow Size_p := Size_{A_p}\)

**Lemma 8.** Assuming an underlying spanning tree and the root knowing the network size, Algorithm PSB is silent and stabilizes in at most \(h\) rounds.

5.2. Algorithm PN

Algorithm PN, shown as Algorithm 5.7, uses the weight \((Weight)\) of the nodes in a tree to assign the node labels. Every processor \(p\) maintains an array \(Start_p\), which holds the starting index for each descendant of \(p\).

The predicate \(InvStart\) is true if a node \(p\) detects that any index assigned to any of its descendants is inconsistent. If there exists any such inconsistent descendant \(q, p\) corrects \(Start_p[q]\) variable by using the macro \(Distribute_p\). This macro assigns names to all the descendant nodes of node \(p\) according to
its own name and the weight of its subtrees. The predicate \( \text{InvNodeLabel} \) is true when a node detects that its name variable \((\eta)\) is incorrect, i.e., different from the one assigned by its parent.

**Algorithm 5.7 \((PN)\)** Parallel Naming using Spanning Tree and Weight Computation.

**Uses**
- For the root \((p = r)\) \: \(D_p\) : set of integer;
- For the leaf processors \: \(A_p, Weight_p\) : integer;
- For the internal processors \: \(D_p, A_p, Weight_p\) : integer

**Constants**
- For the root \((p = r)\) \: \(\eta_p = 0\)

**Variables**
- For the root \((p = r)\) \: \(\text{given}_p\) : integer;
- \(\text{Start}_p[1..|D_p|]\) : array of integer;
- For the leaf processors \: \(\eta_p\) : integer;
- \(\text{given}_p\) : integer;
- \(\text{Start}_p[1..|D_p|]\) : array of integer

**Predicates**
- \(\text{InvNodeLabel}(p) \equiv \eta_p \neq \text{Start}_p[p]\)
- \(\text{InvStart}(p) \equiv (\text{Start}_p[1] \neq \eta_p + 1) \lor (\exists q \in D_p \setminus \{1\} :: \text{Start}_p[q] \neq \text{Start}_p[q - 1] + \text{Weight}_{q-1})\)

**Macros**
\[
\text{Distribute}_p = \begin{cases} 
\text{given}_p := \eta_p; \\
\text{for } q := 1 \text{ to } |D_p| \text{ do} \\
\text{Start}_p[q] := \text{given}_p + 1; \text{given}_p := \text{given}_p + \text{Weight}_q 
\end{cases}
\]

**Actions**
- \(\{\text{For the internal processors}\}\)
  \(\text{InvNodeLabel}(p) \lor \text{InvStart}(p) \rightarrow \eta_p := \text{Start}_p[p]; \text{Distribute}_p\)

- \(\{\text{For the root}\}\)
  \(\text{Distribute}_p\)

- \(\{\text{For the leaf processors}\}\)
  \(\text{InvNodeLabel}(p) \rightarrow \eta_p := \text{Start}_p[p]\)

The composition of node labeling and weight computation is shown in Figure 5.2. The broken lines in the trees in Figure 5.2 indicate the edges in the original graph which were not chosen in the spanning tree by Algorithm \(ST\). We start from a configuration where the variables can have any value (represented by a “?”), except \(\text{Weight}\) of the leaf processors is always 1 and \(\eta\) of the root is 0 (Figure 5.2 (i)). The parent of a leaf node, i.e., an internal
node, corrects its Weight variable (Figure 5.2 (ii)). This propagates to the root, which does a similar computation to correct its Weight variable (see Algorithm PWC) (Figure 5.2(iii)). In this configuration, for each node \( p \in V \), the \( Weight_p \) reflects the actual weight of the subtree rooted at node \( p \). Now, using the weight of each of its subtrees (Distribute macro) and its own name, the root assigns correct name to its descendants in variable \( Start_r \) (1, 4, 5 in Figure 5.2(iii)). Each descendant chooses the name assigned by the root, and assigns the name to its descendants in a similar manner (Figure 5.2 (iv)). Thus, when all the leaves have been assigned a name, all nodes in the network have a unique \( \eta \in \{0, \ldots, n-1\} \) (Figure 5.2 (v)).

Lemma 9. Assuming an underlying spanning tree and the proper assignments of weights of all nodes, Algorithm PN is silent, and stabilizes in at most \( h \) rounds.

5.3. Algorithm NOST

Algorithm NOST is shown as Algorithm 5.8. NOST is a generalized collateral composition of PEL, PN, PSB, PWC, and ST.

We first define the notion of parallel composition.

Definition 7 (Parallel composition). Let \( S_1 \) and \( S_2 \) be programs such that the set of variables (both read and write variables) used in \( S_1 \) and \( S_2 \) are disjoint.
The parallel composition of $S_1$ and $S_2$, denoted by $S_1 || S_2$ or $S_2 || S_1$, is the program that has all the variables and actions of $S_1$ and $S_2$.

**Lemma 10.** $S_2 || S_1$ stabilizes to $L_1 \land L_2$ if the following two conditions hold:
1. $S_1$ stabilizes to $L_1$.
2. $S_2$ stabilizes to $L_2$.

**Algorithm 5.8** ($NOST$) Network Orientation based on Spanning Tree.

\[ P \oplus L \circ (P N \| P S B) \circ P W C \circ ST \]

Let $L_{ST}$ and $L_W$ be the predicates corresponding to the specification of a spanning tree construction and a subtree weight computation, respectively. We first prove that the collateral composition $P W C \circ ST$ is self-stabilizing.

**Lemma 11.** $P W C \circ ST$ is a silent self-stabilizing algorithm for the legitimacy predicate $L_W$.

**Proof.**
1. Algorithm $ST$ stabilizes to $L_{ST}$ (by assumption).
2. Algorithm $P W C$ stabilizes to $L_W$ if $L_{ST}$ holds (Lemma 7).
3. Algorithm $ST$ does not change variables read by $P W C$ once $L_{ST}$ holds.
4. The composition is fair w.r.t. both $P W C$ and $ST$ (every execution contains an infinite suffix in which no steps of $P W C$ are enabled).

By Theorem 1, $P W C \circ ST$ is self-stabilizing, and since $P W C$ and $ST$ are two silent algorithms, it is obvious that the composite algorithm is stabilizing and silent.

Now we prove that the composite algorithm $(P N \| P S B) \circ P W C \circ ST$ is self-stabilizing.

**Lemma 12.** $(P N \| P S B) \circ P W C \circ ST$ is a silent self-stabilizing algorithm for the legitimacy predicate $L_{NL} \land L_S$.

**Proof.**
1. Algorithm $(P N \| P S B) \circ P W C \circ ST$ stabilizes to $L_{ST} \land L_W$ (Lemma 11).
2. The composite algorithm $(P N \| P S B)$ stabilizes to $L_{NL} \land L_S$ if $L_{ST} \land L_W$ holds (Lemmas 8, 9, and 10).
3. Algorithm $(P N \| P S B)$ does not change variables read by $(P N \| P S B)$ once $L_{ST} \land L_W$ holds ($P W C \circ ST$ is silent by Lemma 11).
4. The composition is fair w.r.t. both $(P N \| P S B)$ and $P W C \circ ST$ (every execution contains an infinite suffix in which no step of $(P N \| P S B)$ is enabled).

By Theorem 1, $(P N \| P S B) \circ P W C \circ ST$ is self-stabilizing, and since $(P N \| P S B)$ and $P W C \circ ST$ are two silent algorithms, the composition is stabilizing too and silent.

SIU 2001
The next step is to prove that the collateral composition of $\mathcal{P}E\mathcal{L}$ and the above composite algorithm is self-stabilizing. The proof is similar to that of Theorem 6. So, we can use Theorem 1 to prove the following for Algorithm $\mathcal{NOST}$:

**Theorem 13.** Algorithm $\mathcal{NOST}$ is a silent self-stabilizing algorithm for $\mathcal{S}\mathcal{P}_{N\mathcal{O}}$.

### 5.4. Space and Time Complexity of Algorithm $\mathcal{NOST}$

The time complexity measure for this protocol composition is the same as that of every constituent protocol, $\mathcal{P}\mathcal{N}$, $\mathcal{P}\mathcal{S}\mathcal{B}$, and $\mathcal{P}\mathcal{W}\mathcal{C}$, all of which stabilize in at most $h$ rounds. There exist several spanning tree construction protocols in the literature which construct a breadth-first search tree of height $h$ in time $O(d)$ (the stabilization time). Thus, Algorithm $\mathcal{NOST}$ stabilizes in $O(d)$.

Algorithms $\mathcal{P}\mathcal{N}$, $\mathcal{P}\mathcal{S}\mathcal{B}$, $\mathcal{P}\mathcal{W}\mathcal{C}$, and $\mathcal{S}\mathcal{T}$ require a total extra space of $O((\Delta_{\text{Tree}} \times \log n) + \Delta_p)$ bits per processor, where $\Delta_{\text{Tree}}$ is the number of neighbors of $p$ in the underlying spanning tree. So, the total space requirement for $\mathcal{NOST}$ is asymptotically the same as that of Algorithm $\mathcal{NOST}$, which is $O(\Delta_p \times \log n)$.

### 6. Conclusions

We proposed the first self-stabilizing network orientation algorithms. All the protocols in this paper set up a chordal sense of direction in the network.

The first algorithm is a very simple orientation protocol, called $\mathcal{P}\mathcal{E}\mathcal{L}$. Algorithm $\mathcal{P}\mathcal{E}\mathcal{L}$ assumes that every processor knows the size ($n$) of the network and the processors are already assigned valid node labels. This protocol stabilizes in only 1 round and is silent.

For the other two protocols, we consider any arbitrary rooted network. Thus, they are applicable to a wide range of network structures. We used different protocol composition techniques to simplify the presentation and proofs. Algorithm $\mathcal{NOST}$ is the composition of a depth-first token circulation protocol, a naming and size computation algorithm, and $\mathcal{P}\mathcal{E}\mathcal{L}$. The algorithm stabilizes in $O(n \times d)$. Algorithm $\mathcal{NOST}$ is the composition of five protocols. All the protocols involved in this composition are silent. Hence, Algorithm $\mathcal{NOST}$ is silent and stabilizes in $O(d)$ using any breadth-first tree construction protocol (e.g., [12]).
References


SIU 2001

Authors addresses:

Ajoy K. Datta. Department of Computer Science, University of Nevada at Las Vegas
Las Vegas, NV 89154-4019, USA
Email. datta@cs.unlv.edu

Shivashankar Gurumurthy
Department of Computer Science, University of Nevada Las Vegas
Las Vegas, NV 89154-4019, USA

Franck Petit
LaRIA, Université de Picardie Jules Verne
5, rue du Moulin Neuf, 80000 Amiens, France
Email. petit@laria.u-picardie.fr

Vincent Villain
LaRIA, Université de Picardie Jules Verne
5, rue du Moulin Neuf, 80000 Amiens, France
Email. villain@laria.u-picardie.fr

SIU 2001