Self-Stabilizing Network Orientation Algorithms In Arbitrary Rooted Networks

Ajoy K. Datta,[1*] Shivashankar Gurumurthy,[1] Franck Petit,[2]
Vincent Villain[2†]
[1] Department of Computer Science, University of Nevada Las Vegas
[2] LaRIA, Université de Picardie Jules Verne, France

Abstract

This paper presents the first deterministic self-stabilizing network orientation algorithms. We present three protocols for arbitrary and asynchronous 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 edge labels (assigned to the links) are valid (meaning, they satisfy the specification of the orientation problem).

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 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 of each processor 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 [13] 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 where no orientation is available. Korfhage and Gafni [12] presented an algorithm to orient directed tori. The orientation problem for tori was also studied by Syrotiuk et al. [14]. There has also been considerable interest in the problem of orienting a ring network [3, 11, 15]. A major work in the area of orientation has been reported in [17], which reported studies on orientation of cliques, hypercubes, and tori in both anonymous and non-anonymous networks.

Another related area of research is the Sense Of Direction (SoD) [8], which allows processors to communicate efficiently, by exploiting the topological properties of the network algorithmically. Flocchini, et. al. [8] 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 [18] 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.

Contributions. In this paper, we propose the first self-stabilizing protocol for orienting an arbitrary rooted network. The first algorithm ($\mathcal{PEL}$) assumes pre-assigned node labels 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 Δ_p is the degree of p.

Next we assume only rooted networks and present

*Supported in part by a sabbatical leave grant from University of Nevada, Las Vegas.
†Supported in part by the Pôle de Modélisation de Picardie, France.

and size of the network. The first scheme, called $\mathcal{SNSC}$, uses a token to implement a sequential naming and size computation protocol. We assume that an underlying depth-first token circulation protocol [5] is maintained on an arbitrary network. We use the protocol proposed in [5] because this protocol has the best space complexity among the known algorithms and, once stabilized, in this protocol, the token follows 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). The protocol does not require an underlying tree to be maintained. Using Algorithm $\mathcal{PEL}$ 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 $\mathcal{SNSC}$ requires only $O(\log \Delta_p + \log n)$ bits per processor.

The second scheme is based on a composition of three algorithms: a parallel weight computation ($\mathcal{PWC}$), a parallel naming ($\mathcal{PN}$), and a parallel size broadcast ($\mathcal{PSB}$). We obtain an orientation protocol, called $\mathcal{NOST}$, by composing $\mathcal{PEL}$, $\mathcal{PN}$, $\mathcal{PSB}$, $\mathcal{PWC}$, and a self-stabilizing spanning tree construction. Using a self-stabilizing spanning tree construction algorithm, (e.g., [1, 2, 7, 10]) this network orientation protocol stabilizes in $O(d)$ rounds where d is the diameter of the network. Algorithms $\mathcal{PN}$, $\mathcal{PSB}$, $\mathcal{PWC}$, and self-stabilizing spanning tree construction algorithm require an extra space of $O((\Delta^{Tree}{}_p \times \log n) + \Delta_p)$ bits per processor, where $\Delta^{Tree}{}_p$ is the number of neighbors of p in the underlying spanning tree.

Outline of the Paper. In Section 2, we present the model of the distributed system considered in this paper. We also introduce the notion of self-stabilization and the problem of network orientation there. In Section 3, we present the self-stabilizing $\mathcal{PEL}$. The other two orientation protocols are presented in Sections 4 and 5. (The full correctness proofs for algorithms in Sections 3, 4, and 5 are provided in [4].) 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 proof of our algorithms. We then define the problem of network orientation in arbitrary rooted networks.

System and Programs. 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 $\mathcal{N}_p$. We assume that $\mathcal{N}_p$ is a constant and is maintained by an underlying protocol. The *degree* of p is denoted by Δ_p and is equal to $|\mathcal{N}_p|$.

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.

Each action is uniquely identified by a label $\mathcal{A}$ and is of the following form: $< \mathcal{A} >:: < \mathcal{G} > \longrightarrow < \mathcal{S} >$. The guard $\mathcal{G}$ 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 $\mathcal{G}$ of an action labeled $\mathcal{A}$ in the program of p is true, then the processor p and the action $\mathcal{A}$ are said to be *enabled*. The statement $\mathcal{S}$ 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 *configuration* of a system is a product of the states of all processors ($\in V$). Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, 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 $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration. *Maximality* 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 an *unfair and distributed daemon* in Sec-

in Section 4. 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 *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. The *distributed* daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses a nonempty subset of these enabled 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.

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. $\forall \alpha$ such that $\mathcal{L}_{\mathcal{P}}$ holds, $\forall e \in \mathcal{E}_\alpha :: e$ satisfies $\mathcal{SP}_{\mathcal{P}}$.
2. $\forall e \in \mathcal{E}$, there exists a suffix of e which satisfies $\mathcal{SP}_{\mathcal{P}}$ (closure and convergence).

2.2 Protocol Composition

The following definitions and result are from [9, 16].

Definition 2.1 (Collateral composition) *Let S_1 and S_2 be programs such that no variable written by S_2 occurs 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 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 state what it means for a composite algorithm to be self-stabilizing.

Definition 2.2 (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.*

$S_2 \circ S_1$ is fair w.r.t. S_i ($i \in \{1, 2\}$) if any execution of $S_2 \circ S_1$ is fair w.r.t. S_i.

Theorem 2.1 *$S_2 \circ S_1$ stabilizes to $\mathcal{L}_2$, if the following four conditions hold:*
1. *S_1 stabilizes to $\mathcal{L}_1$.*
2. *S_2 stabilizes to $\mathcal{L}_2$ if $\mathcal{L}_1$ holds.*
3. *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 given in [8]. 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 $\psi : V \longrightarrow V$ 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 $\delta : V \times V \longrightarrow \{0, \ldots, n-1\}$ be the corresponding distance function, i.e., $\delta(p, q)$ is the smallest k such that $\psi^k(p) = q$. The labeling π is a chordal labeling iff, $\forall (p, q) \in E_p :: \pi_p(p, q) = \delta(p, q)$.

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 η_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:
($SP1$) Every node in the network has a unique (permanent) name η_p in the range $0, \ldots, n-1$.
($SP2$) $\forall p \in V : \forall l \in E_{p,q} :: \pi_p[l] = (\eta_p - \eta_q)\ mod\ n$.

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 present Algorithm $\mathcal{PEL}$ and the sketch of the correctness proof. It is followed by a brief analysis of the space and time complexity.

The variable π_p is an array that stores the edge label for every edge incident on p. The algorithm uses variables η_p and S_p as constants (they can be computed using the naming and size computation techniques as discussed later). The algorithm is very simple and consists of a single guarded statement, which checks all the edge labels for correctness, using the predicate $InvElab(p)$. The macro $Edgelabel_p$ computes the edge label.

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

Uses: $S_p,\ \eta_p$: integer;
Variables: $\pi_p[1..|\mathcal{N}_p|]$: array of integer;
Predicates: $InvElab(p) \equiv$
$\{\exists q \in \mathcal{N}_p :: \pi_p[E_{p,q}] \neq (\eta_p - \eta_q)\ mod\ S_p\}$
Macro: $Edgelabel_p =$
for all $q \in \mathcal{N}_p$ **do**
$\pi_p[E_{p,q}] := (\eta_p - \eta_q)\ mod\ S_p;$
Actions: $InvElab(p) \longrightarrow Edgelabel_p;$

We define the predicates $\mathcal{L}_{\mathcal{NL}}$, $\mathcal{L}_{\mathcal{S}}$, and $\mathcal{L}_{\mathcal{EL}}$ as follows:
1. $\mathcal{L}_{\mathcal{NL}} \equiv$ Every node in the network has a unique (permanent) name η_p in the range $0, \ldots, n-1$.
2. $\mathcal{L}_{\mathcal{S}} \equiv \forall p \in V :: S_p = n$.
3. $\mathcal{L}_{\mathcal{EL}} \equiv \forall p \in V : \forall l \in E_{p,q} :: \pi_p[l] = (\eta_p - \eta_q)\ mod\ S_p$.

In this section, for every processor p, we assume name η_p as a constant and unique for each p. Also, every processor p has the knowledge of the size of the network (S_p). So, $\forall \alpha \in \mathcal{C} :: \mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ holds. Furthermore, no processor can execute more than one action. Since η_p and S_p are constants, any processor p, enabled to execute its action, is continuously enabled until p executes the action. But, once p executes its action, p will not be enabled forever. So, every p cannot execute its action more than once. This leads to the following result:

Theorem 3.1 *Algorithm $\mathcal{PEL}$ is a silent self-stabilizing algorithm even if the daemon is distributed and unfair.*

3.2 Space and Time Complexity

The time to stabilize Algorithm $\mathcal{PEL}$ is at most 1 round, since all enabled processors can move independently. Every node maintains one array variable, π_p, which has Δ_p elements. Every element of the array is in $1, \ldots, n-1$. Thus the space complexity of Algorithm $\mathcal{PEL}$ is $O(\Delta_p \times \log n)$ bits per processor.

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 $\mathcal{SNSC}$. We then define a new protocol composition, called *conditional composition*, and show that the conditional composite algorithm $\mathcal{SNSCTC}$ (Sequential Naming, Size Computation, and depth-first Token Circulation) is self-stabilizing. Finally, we present Algorithm $\mathcal{NOTC}$ (Network Orientation based on Token Circulation), using the collateral composition of $\mathcal{SNSCTC}$ and $\mathcal{PEL}$, and its correctness proof.

4.1 Algorithm $\mathcal{SNSC}$

We use the depth-first token circulation algorithm of [5], which we call Algorithm $\mathcal{DFTC}$. Algorithm $\mathcal{SNSC}$ reads two variables of $\mathcal{DFTC}$: 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 Cnt_p, S_p, and η_p. Cnt_p is used to count the number of nodes visited by the token. S_p refers to the number of processors in the network, i.e., the maximum value of Cnt_p in the whole network. η_p maintains the label of the node corresponding to processor p.

Algorithm $\mathcal{SNSC}$ is shown as Algorithm 4.2. The macro $UpdCnt_p$, $UpdSize_p$, and $Nodelabel_p$ is used to update Cnt_p, to update the size of the network, and to name the node, respectively.

A node is said to hold a token if the following predicate holds: $Token(p) \equiv Forward(p) \vee Backtrack(p)$, where $Forward(p)$ and $Backtrack(p)$ are predicates from Algorithm $\mathcal{DFTC}$. $Forward(p)$ is enabled at processor p, when it receives a token for the first time from its parent A_p. The predicate $Backtrack(p)$ is true every time the token is backtracked to processor p from its descendant D_p. For a more detailed description of $Forward(p)$ and $Backtrack(p)$, refer to [5].

The depth-first token circulation protocol guarantees that every node, during a single round, will have its $Forward(p)$ enabled exactly once, i.e., it will have the token at least once. When a node has a token for the

Computation using Depth-First Token Circulation.

Uses:

For every p: D_p : integer;
$Forward(p)$, $Backtrack(p)$: predicates;
For every $p \neq r$: A_p : integer ;
Constants: For $p = r$ :: $\eta_p = 0$;
Variables:
For every p: S_p, Cnt_p : integer;
For every $p \neq r$: η_p : integer;
Macro:

$$UpdSize_p \equiv \begin{cases} S_p := Cnt_p; Cnt_p := 0; & \textbf{if } (p = r) \\ S_p := S_{A_p}; Cnt_p := Cnt_{A_p} + 1; & \textbf{otherwise}; \end{cases}$$

$Nodelabel_p \equiv$ $\eta_p := Cnt_p$; **if** $(p \neq r)$;
$UpdCnt_p \equiv$ $Cnt_p := Cnt_{D_p}$;
Actions:
$Forward(p)$ $\longrightarrow UpdSize_p$; $Nodelabel_p$;
$Backtrack(p)$ $\longrightarrow UpdCnt_p$;

first time, it assigns the next lowest available name, as its node label, after consulting its parent. The node then passes the token to the next node (descendant), if any. Otherwise, it backtracks the token to its parent along with the current Cnt value.

4.2 Algorithm $\mathcal{SNSCTC}$

We use a new protocol composition technique, called *conditional composition* to simplify the design and proof for the $\mathcal{SNSCTC}$ algorithm. We first define the notion of *conditional composition*.

Definition 4.1 (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 g of S_2 has the form $g_1 \wedge g_2$ or $\neg(g_1) \wedge g_2$ where g_1 is a logical expression using the guards $\in \mathcal{G}$.*
4. *If actions of both S_1 and S_2 are enabled in the same step, action of S_2 is followed by action of S_1.*

Algorithm $\mathcal{SNSCTC}$ is shown as Algorithm 4.3.

We now explain the order of execution of the actions of Algorithm $\mathcal{SNSCTC}$. The first action of $\mathcal{SNSC}$ is guarded by $Forward$ predicate. So, when this predicate is true in $\mathcal{SNSCTC}$, the processor first executes

Computation, and Depth-First Token Circulation.

$\mathcal{SNSC} \circ |_{\{Forward, Backtrack\}} \mathcal{DFTC}$

$UpdSize_p$ and $Nodelabel_p$, then the action in the Algorithm $\mathcal{DFTC}$ corresponding to the predicate $Forward$ (it sends the token to its first descendant, if any, or to its ancestor, otherwise). Since Algorithm $\mathcal{DFTC}$ [5] stabilizes (we use $\mathcal{L}_{\mathcal{TC}}$ to denote the corresponding legitimate predicate) only if the daemon is *weakly fair* (or stronger), we now assume in this section that the daemon is distributed and weakly fair.

We cannot use Theorem 2.1 directly to prove the correctness of Algorithm $\mathcal{SNSCTC}$ since we used the conditional composition here. But, we can make the following observations:
1. Algorithm $\mathcal{DFTC}$ stabilizes to $\mathcal{L}_{\mathcal{TC}}$. (Algorithm $\mathcal{SNSC}$ has no impact on the behavior of $\mathcal{DFTC}$.)
2. Algorithm $\mathcal{SNSC}$ stabilizes to $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ if $\mathcal{L}_{\mathcal{TC}}$ holds. (Once stabilized, the token follows the same path forever. Thus, the node labels assigned are permanent after the first normal token circulation, and at the end of the next token circulation, every processor knows the size of the network.)
3. Algorithm $\mathcal{DFTC}$ **does** change variables read by $\mathcal{SNSC}$ after $\mathcal{L}_{\mathcal{TC}}$ holds. So, the third point of Theorem 2.1 is not satisfied. But, once Algorithm $\mathcal{DFTC}$ stabilizes, (as discussed before) the token always takes the same route. Hence, the size and the node naming are eventually permanent. Thus, the variables read by $\mathcal{SNSC}$ after $\mathcal{L}_{\mathcal{TC}}$ holds, effectively do not change.
4. The composition is fair w.r.t. both $\mathcal{DFTC}$ and $\mathcal{SNSC}$. (Every time $Forward$ or $Backtrack$ is true, action of both $\mathcal{DFTC}$ and $\mathcal{SNSC}$ is executed.)

So, we can claim the following result:

Lemma 4.1 *Algorithm $\mathcal{SNSCTC}$ stabilizes for the specification defined by the legitimacy predicate $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ if the distributed daemon is weakly fair or stronger.*

4.3 Space and Time Complexity of Algorithm $\mathcal{SNSCTC}$

The time to stabilize Algorithm $\mathcal{SNSCTC}$ is $O(n \times d)$. 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.

on Token Circulation.

$\mathcal{PEL} \circ \mathcal{SNSCTC}$

4.4 Algorithm $\mathcal{NOTC}$

Algorithm $\mathcal{NOTC}$ (shown as Algorithm 4.4) orients any rooted network. It is a collateral composite algorithm. We can make the following observations:
1. Algorithm $\mathcal{SNSCTC}$ stabilizes to $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ (see Lemma 4.1).
2. Algorithm $\mathcal{PEL}$ stabilizes to $\mathcal{L}_{\mathcal{EL}}$ if $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ holds (see Theorem 3.1).
3. Algorithm $\mathcal{SNSCTC}$ does not change variables read by $\mathcal{PEL}$ once $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ holds.
4. The composition is fair w.r.t. both $\mathcal{SNSCTC}$ and $\mathcal{PEL}$ (every execution contains an infinite suffix in which no step of $\mathcal{PEL}$ is enabled).

Then, by Theorem 2.1, we obtain:

Theorem 4.1 *Algorithm $\mathcal{NOTC}$ is self-stabilizing for $\mathcal{SP}_{\mathcal{NO}}$ if the distributed daemon is weakly fair or stronger.*

It is easy to notice that the above result would also hold for an unfair daemon if one uses a DFTC scheme which stabilizes under the same (unfair) daemon.

4.5 Space and Time Complexity of Algorithm $\mathcal{NOTC}$

Since the stabilizing time of $\mathcal{PEL}$ is 1, the time complexity measure for this protocol composition is the same as that of $\mathcal{SNSCTC}$, 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 $\mathcal{SNSCTC}$ and $O(\Delta_p \times \log n)$ for $\mathcal{PEL}$).

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 $\mathcal{ST}$) to orient the network. We assume that Algorithm $\mathcal{ST}$ deterministically maintains a spanning tree of the graph. In $\mathcal{ST}$, 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 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 $\mathcal{PWC}$) and a size broadcast algorithm (Algorithm $\mathcal{PSB}$). We then give the parallel naming algorithm (Algorithm $\mathcal{PN}$). Finally, we present the algorithm $\mathcal{NOST}$ (Network Orientation using Spanning Tree protocol) which is a composition of $\mathcal{PEL}$, $\mathcal{PN}$, $\mathcal{PSB}$, $\mathcal{PWC}$, and $\mathcal{ST}$.

5.1 Algorithms $\mathcal{PWC}$ and $\mathcal{PSB}$

Algorithm $\mathcal{PWC}$. The variable W_p maintains the size (or the number of nodes) of the subtree rooted at p. The predicate $InvW$ becomes true if an internal node or the root detects that its weight is incorrect. In that situation, the node computes W_p 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 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 to the right value. The implementation of $\mathcal{PWC}$ is given in Algorithm 5.5.

Algorithm 5.5 $(\mathcal{PWC})$ Parallel Weight Computation using Spanning Tree.

Uses:
For $p = r$ and every internal p: $\mathcal{D}_p$: set of integer;
Constants:
For every leaf p: $W_p = 1$;
Variables:
For $p = r$ and every internal p: W_p : integer;
Predicates:
$InvW(p) \equiv W_p \neq 1 + \sum_{q \in \mathcal{D}_p} W_q$;
Actions:
For $p = r$ and every internal p:
$InvW(p) \longrightarrow W_p := 1 + \sum_{q \in \mathcal{D}_p} W_q$;

The following lemma is obvious:

Lemma 5.1 *Assuming an underlying spanning tree, Algorithm $\mathcal{PWC}$ is silent, and stabilizes in at most h steps even if the daemon is unfair.*

Algorithm $\mathcal{PSB}$. 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.

Spanning Tree and Weight Computation.

Uses:

For $p = r$: W_p : integer;
For every leaf and internal p: A_p : integer;

Constants:

For $p = r$: $S_p = W_p$;

Variables:

For every leaf and internal p: S_p : integer;

Actions:

For every leaf and internal p:
$S_p \neq S_{A_p} \longrightarrow S_p := S_{A_p}$;

Lemma 5.2 *Assuming an underlying spanning tree and the root knowing the network size, Algorithm $\mathcal{PSB}$ is silent, and stabilizes in at most h steps even if the daemon is unfair.*

5.2 Algorithm $\mathcal{PN}$

Algorithm $\mathcal{PN}$, shown as Algorithm 5.7, uses the weight of the nodes in a tree to assign the node labels. Every processor maintains an array $Start$, which holds the starting index for each descendant for the processor.

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 the corresponding $Start$ 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 $InvNlab$ is true when a node detects that its name variable (η) is incorrect, i.e., different from the one assigned by its parent.

Lemma 5.3 *Assuming an underlying spanning tree and the proper assignments of weights of all nodes, Algorithm $\mathcal{PN}$ is* silent, *and stabilizes in at most h steps even if the daemon is unfair.*

5.3 Algorithm $\mathcal{NOST}$

Algorithm $\mathcal{NOST}$ is shown as Algorithm 5.8. $\mathcal{NOST}$ is a generalized composition of $\mathcal{PEL}$, $\mathcal{PN}$, $\mathcal{PSB}$, $\mathcal{PWC}$, and $\mathcal{ST}$.

We first define the notion of *parallel composition*.

Definition 5.1 (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.*

ning Tree and Weight Computation.

Uses:

For $p = r$: $\mathcal{D}_p$: set of integer;
For every leaf$p$: A_p, W_p : integer;
For every internal p: $\mathcal{D}_p$: set of integer; A_p, W_p : integer;

Constants:

For $p = r$: $\eta_p = 0$;

Variables:

For $p = r$: $given_p$: integer;
$Start_p[1..|\mathcal{D}_p|]$: array of integer;
For every leaf p: η_p : integer;
For every internal p: $\eta_p, given_p$: integer;
$Start_p[1..|\mathcal{D}_p|]$: array of integer;

Predicates:

$InvNlab(p) \equiv \eta_p \neq Start_{A_p}[p]$
$InvStart(p) \equiv$
$(Start_p[1] \neq \eta_p + 1)$
$\vee\ (\exists q \in \mathcal{D}_p \setminus \{1\} :: Start_p[q] \neq Start_p[q-1] + W_{q-1})$

Macros:

$Distribute_p \equiv$
$given_p := \eta_p$;
for $q := 1$ **to** $|\mathcal{D}_p|$ **do**
$Start_p[q] := given_p + 1;\ given_p := given_p + W_q$;

Actions:

For every internal p:
$InvNlab(p) \vee InvStart(p) \longrightarrow$
$\eta_p := Start_{A_p}[p];\ Distribute_p$;
For $p = r$:
$InvStart(p) \longrightarrow Distribute_p$;
For the leaf p:
$InvNlab(p) \longrightarrow \eta_p := Start_{A_p}[p]$;

Lemma 5.4 *$S_2||S_1$ stabilizes to $\mathcal{L}_1 \wedge \mathcal{L}_2$ if the following two conditions hold:*
1. *S_1 stabilizes to $\mathcal{L}_1$.*
2. *S_2 stabilizes to $\mathcal{L}_2$.*

Algorithm 5.8 $(\mathcal{NOST})$ Network Orientation based on Spanning Tree.

$\mathcal{PEL} \circ (\mathcal{PN}||\mathcal{PSB}) \circ \mathcal{PWC} \circ \mathcal{ST}$

Let $\mathcal{L}_{\mathcal{ST}}$ and $\mathcal{L}_{\mathcal{W}}$ be the predicates corresponding to the specification of a spanning tree construction and a subtree weight computation, respectively.

Theorems 2.1, 3.1, and Lemmas 5.1 to 5.4 lead to the following results:

rithm for the legitimacy predicate $\mathcal{L}_{\mathcal{W}}$ even if the daemon is unfair.

Lemma 5.6 *$(\mathcal{PN}||\mathcal{PSB}) \circ \mathcal{PWC} \circ \mathcal{ST}$ is a silent self-stabilizing algorithm for the legitimacy predicate $\mathcal{L}_{\mathcal{NL}} \wedge \mathcal{L}_{\mathcal{S}}$ even if the daemon is unfair.*

Theorem 5.1 *Algorithm $\mathcal{NOST}$ is a silent self-stabilizing algorithm for $\mathcal{SP}_{\mathcal{NO}}$ even if the daemon is unfair.*

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

The time complexity of this protocol composition is the same as that of every constituent protocol, $\mathcal{PN}$, $\mathcal{PSB}$, and $\mathcal{PWC}$, all of which stabilize in at most h steps. 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 $\mathcal{PN}$, $\mathcal{PSB}$, $\mathcal{PWC}$, and $\mathcal{ST}$ require a total extra space of $O((\Delta^{Tree}{}_p \times \log n) + \Delta_p)$ bits per processor, where $\Delta^{Tree}{}_p$ 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 $\mathcal{NOTC}$, which is $O(\Delta_p \times \log n)$.

6 Conclusions

We proposed the first deterministic self-stabilizing network orientation algorithm. 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 $\mathcal{PEL}$. Algorithm $\mathcal{PEL}$ 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 step and is silent, even if the daemon is unfair. For the other two protocols, we consider any arbitrary rooted network. We used different protocol composition techniques: *Collateral composition* as presented in [16], and *conditional* and *parallel composition* as defined in this paper. The composition approach made the design and proof simpler. Algorithm $\mathcal{NOTC}$ is based on a depth-first token circulation which runs using a weakly fair daemon, and stabilizes in $O(n \times d\})$. Algorithm $\mathcal{NOST}$ is the composition of five protocols. This algorithm is silent and stabilizes with an unfair daemon, in $O(d)$ using any breadth-first tree construction protocol.

[1] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[2] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652–661, 1993.

[3] I. Cidon and Y. Shavitt. Message terminate algorithms for anonymous rings of unknown size. In *6th International Workshop on Distributed Algorithms, Haifa*, 1992.

[4] A. Datta, S. Gurumurthy, F. Petit, and V. Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. Technical Report 99-24, LaRIA, University of Picardie Jules Verne, 1999.

[5] A. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *SIROCCO'98, The 5th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 229–243, 1998.

[6] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[7] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[8] P. Flocchini, B. Mans, and N. Santoro. Sense of direction: Definitions, properties and classes. In *Tech. Rep. TR-95-10, School of Computer Science, Carleton University, Ottawa*, 1995.

[9] T. Herman. *Adaptativity through Distributed Convergence*. PhD thesis, Dept. of Comp. Science, University of Texas, Austin, 1991.

[10] S. Huang and N. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[11] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[12] W. Korfhage and E. Gafni. Orienting unidirectional torus network. *Manuscript*, 1985.

[13] N. Santoro. Sense of direction, topological awareness, and communication complexity. *ACM SIGACT News*, 16:50–56, 1984.

[14] V. Syrotiuk, C. Colbourn, and J. Pachl. Wang tilings and distributed orientation on anonymous torus networks. In *Proceedings of the 7th International Workshop on Distributed Algorithms, Lausanne*, 1993.

[15] V. Syrotiuk and J. Pachl. A distributed ring orientation problem. In *Proceedings of the 2nd International Workshop on Distributed Algorithms, Amsterdam*, 1987.

[16] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1994.

[17] G. Tel. Network orientation. *International Journal of Foundations of Computer Science*, 5:23–57, 1994.

[18] G. Tel. Sense of direction in processor networks. Technical report, SOFSEM95, LNCS 1012, 1995.