Self-stabilization with Global Rooted Synchronizers
Luc Onana Alima, Joffroy Beauquier, Ajoy Datta, Sébastien Tixeuil

To cite this version:

Self-stabilization with Global Rooted Synchronizers

Luc O. Alima\textsuperscript{1} Joffroy Beauquier\textsuperscript{2} Ajoy K. Datta\textsuperscript{3} Sébastien Tixeuil\textsuperscript{2}

\textsuperscript{1} Unité d’Informatique, Université catholique de Louvain, Belgium
\textsuperscript{2} L.R.I./C.N.R.S., Université de Paris-Sud, France
\textsuperscript{3} Dept of Computer Science, University of Nevada Las Vegas

The final version of this paper appears in [2].

Abstract

We propose a self-stabilizing synchronization technique, called the Global Rooted Synchronization, that synchronizes processors in a tree network. This synchronizer can be used as a compiler to convert a synchronous protocol $P$ for tree networks into a self-stabilizing version of $P$ for tree networks. The synchronizer requires only $O(1)$ memory (other than the memory needed to maintain the tree) at each node regardless of the size of the network, stabilizes in $O(h)$ time, where $h$ is the height of the tree, and does not invoke any global operations. Several applications of this technique are presented, such as the distributed clock synchronization and global calculus computation—each being efficient both in time and space complexity compared to the previous results.

Keywords: Distributed algorithms, self-stabilization, synchronization.

1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient faults because they are exposed to constant change of their environment. The concept of self-stabilization [12] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

In distributed systems, the task of synchronization is crucial, and has been covered in depth in [16], [17], and [18]. Informally, a synchronizer is a protocol that allows asynchronous systems to simulate the behavior of synchronous systems. In many distributed algorithms, the waves of communication (repeated phases of broadcast and convergecast) is often used in order to achieve synchronization among processors (or processes)\textsuperscript{1}.

Related Work. The research in the area of synchronizers started from the seminal work of Awerbuch [5]. After the original paper on self-stabilization by Dijkstra [12], a few general schemes to self-stabilize the non-stabilized protocols have been proposed. The reset protocol in [4] stabilizes in time $O(D^2)$, where $D$ is the upper bound on the diameter of the network. The spanning tree construction algorithm in [1] takes time $O(n^2)$ to stabilize, where $n$ is the

\textsuperscript{1}We assume that every processor runs exactly one process.
number of nodes in the network. In [7], a general method of network reset based on the local checking and local correction is introduced. This method is applicable to interactive protocols and takes time $O(n)$ to stabilize.

Any of the aforementioned protocols could be used along with the synchronizer of [5] to design the self-stabilizing synchronizers. A general self-stabilizing synchronizer applicable to non-interactive protocols is described in [8]. This synchronizer takes $O(D)$ time to stabilize and no extra space other than that required by the original non-stabilizing protocol. The synchronizer in [6] stabilizes in optimal time, i.e., $O(d)$, but takes $O(\log n \log D)$ space, where $d$ is the actual diameter of the network. The counter flushing technique introduced in [19] provides a general scheme for the global synchronization in tree networks. This scheme takes $O(h)$ to stabilize and takes $(\log n)$ extra space to implement the counters, where $h$ is the height of the tree.

**Our Contributions.** In this paper, we present a self-stabilizing synchronizer for tree networks. After some general transient fault has occurred and left the whole system in an arbitrary configuration, the system is allowed to exhibit some “faulty” (i.e., non-synchronized) behavior for a finite time only, but it must converge to a proper execution when all processors are synchronized. Our synchronizer protocol takes only $O(1)$ extra space and stabilizes in $O(h)$ time. We show that several distributed algorithms using this synchronizer is both time and space efficient as compared with the similar existing algorithms.

The self-stabilizing spanning tree construction algorithms have been proposed in [1], [4], [6], [10], and [14]. Any of these algorithms can be combined with our synchronizer to design a synchronizer for a general network.

**Outline of the Paper.** In Section 2, we describe the distributed systems and the model in which our synchronizer is written. In Section 3, we give a formal statement of the synchronizer solved in this paper and also present the synchronizer protocol. Some applications of our synchronizer are presented in Section 4. Finally, we give the concluding remarks in Section 7. In appendix 5, we give the proof of correctness of the protocol. The time and space complexity of the protocol is given in Appendix 6.

## 2 Model

In this section, we define the distributed systems, actions, and computations considered in this paper, and state what it means for a protocol to be self-stabilizing.

**Distributed System.** A distributed system is an undirected, connected graph $S = (V, E)$ where $V$ is the set of nodes ($|V| = n$) and $E$ is the set of links or edges.

A link connecting node $i$ to node $j$ is uniquely identified by the two-tuple $(i, j)$, and for every $(i, j) \in E$, nodes $i$ and $j$ are called neighbors. We consider asynchronous distributed systems in this paper.

We use only tree networks in this paper. If $S$ is a tree based distributed system, the root node is denoted by $S^r$, the set of leaf nodes by $S^l$, and the set of other nodes (called hereafter the intermediate nodes) by $S^o$. So, the set of all nodes, $\{S^r\} \cup S^o \cup S^l = V$. The parent of node $i$ is denoted by $P^i$ and the set of children of node $i$ by $C^i$. 
The height of a particular node \( i \) (i.e., its distance from the root \( S^r \)) is denoted as \( \hat{i} \), and the set of nodes that are up to distance \( d \) from node \( i \) is denoted by \( \mathcal{V}^i_d \). For the sake of convenience, we denote the set of nodes that are up to distance \( d \) from the root as \( \mathcal{V}_d \).

**Communications.** Each processor executes a program and the processors execute their programs asynchronously. The program consists of a set of variables and a finite set of actions. The processors have two types of variables: *local variables* and *field variables*. The field variables are part of the shared register which is used to communicate with the neighbors. The local variables are defined in the program of processor \( i \) and are used strictly locally, meaning that they cannot be accessed by the neighbors of \( i \). A processor can only write to its own shared register and can only read shared registers owned by the neighboring processors. So, the field variables of \( i \) can be accessed by \( i \) and its neighbors. In such a model, each processor \( i \) can access two disjoint sets of registers:

1. \( \text{WR}_i = \{ l_{ij} | (i,j) \in E \} \) is the set of registers in which \( i \) may write information into using the `write` primitive.
2. \( \text{RR}_i = \{ l_{ji} | (i,j) \in E \} \) is the set of registers in which \( i \) may read information from using the `read` primitive.

As a register \( l_{ij} \) serves as a channel of communication from \( i \) to \( j \), in the sequel, \( \text{WR}_i \) will represent \( i \)'s registers. Processor \( i \) performing `read`\( (l_{ji}) \) reads \( l_{ji} \in \text{RR}_i \) (and also \( \in \text{WR}_j \)). Processor \( i \) performing `write`\( (l_{ij}, \text{list-of-values}) \) writes `list-of-values` to the corresponding fields of \( l_{ij} \in \text{WR}_i \) (and also \( \in \text{RR}_j \)).

**States and Configurations.** The *state* of processor is defined by the values of its local variables. The state of a link \( (i,j) \in E \) is defined by the values of \( l_{ij} \) and \( l_{ji} \). A *configuration* of a distributed system \( S = (V, E) \) is an instance of the states of its processors and links. The set of configurations of \( S \) is denoted as \( \mathcal{C} \).

**Actions and Computations.** A processor *action* consists of an internal computation along with one or more `read` or `write` actions. This execution model is known as the composite atomicity model. 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 \( i \) is a boolean expression involving the local variables of \( i \), and the field variables of \( i \) and its neighbors. An action can be executed only if its guard evaluates to true. 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 atomic execution of an action of \( i \) is called a *step* of \( i \). Processor actions change the global system configuration. Moreover, several processor actions may occur at the same time.

A *computation* \( e \) of a protocol \( \mathcal{P} \) is a *fair, maximal* sequence of configurations \( c_1, c_2, \ldots \) such that for \( i = 1, 2, \ldots \), the configuration \( c_{i+1} \) is reached from \( c_i \) by a single step of at least one processor. \( c_1 \) is called the *initial configuration* of \( e \). During a computation step, one or more processors execute a step and a processor may take at most one step. *Fairness* of the sequence means that if any action in \( \mathcal{P} \) is continuously enabled along the sequence, it is
eventually chosen for execution. **Maximality** means that the sequence is either infinite, or it is finite and no action of $P$ is enabled in the final global state. All computations considered in this paper are assumed to be fair and maximal.

The set of computations of a protocol $P$ in system $S$ starting with a particular initial configuration $l \in C$ is denoted by $E_l$. Every computation $e \in E_l$ is of the form $c_1, c_2, \ldots$, with $l = c_1$. The set of computations of $P$ in system $S$ whose initial configurations are all elements of $B \subset C$ is denoted as $E_B$. Thus, $E_C = E$.

**Predicates.** $e \triangleright P$ means that an element $e \in E$ satisfies the predicate $P$ defined on the set $E$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We distinguish two special predicates: **true** and **false**. They are defined as follows: for any $e \in E$, $e \triangleright \text{true}$ and no $e \in E$ matches **false**.

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

**Definition 1 (Attractor)** Let $B_\alpha$ and $B_\beta$ be two predicates of a protocol $P$ defined on the set of configurations, $C$ of system $S$. $B_\alpha$ is an attractor for $B_\beta$ if and only if the following condition is true:

$$\forall b_\beta \triangleright B_\beta : \forall e \in E_{b_\beta} : ((e = c_1, c_2, \ldots) \land b_\beta = c_1) :: \exists i \geq 1, c_i \triangleright B_\alpha.$$

Intuitively, an attractor is a predicate of a protocol $P$ that “attracts” another predicate of $P$ for any computation of $P$ in system $S$.

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

1. $\forall l \triangleright L : \forall e \in E_l :: e \triangleright SP$ (correctness).
2. $L$ is an attractor for **true** (convergence).

The temporal activities of a self-stabilizing protocol can be divided into three phases:

1. The *fault* phase: The period during which faults may occur in the system. These faults may corrupt the volatile memory of processors and links. This phase lasts for $T_f$ time.
2. The *stabilizing* phase: The period during which the system may not exhibit the correct behavior. However, no external faults may occur. This phase lasts for $T_s$ time.
3. The *stabilized* phase: The period during which every computation of the system is correct (i.e., satisfies its specification).

The efficiency of a self-stabilizing protocol can be measured in time and memory needed to achieve the stabilization. Since we consider asynchronous systems, some bound on the communication time must be assumed to find the time complexity of the protocol. We refer this bound as a *time unit*. Any finite sequence of internal actions is assumed to be instantaneous.
Definition 3 (Space complexity) The space complexity of a self-stabilizing protocol is the memory space needed to hold the local and field variables (to maintain the communication information among processors).

Definition 4 (Time complexity) The time complexity of a self-stabilizing protocol is the time needed to reach a configuration that matches the $L$ predicate after the faults cease to occur. This is denoted by $T_s$.

3 Global Rooted Synchronization ($\mathcal{GRS}$) for Tree Networks

In this section, we provide a scheme for stabilizing wave chains of communication on tree networks. We define the following terms in Section 3.1: wave, wave chain, colored wave, and colored wave chain (the color is used to distinguish two consecutive waves). Then we give the formal specification of the global synchronization problem we solve in this paper. We describe, both informally and formally, our solution (Section 3.2).

3.1 Specification of $\mathcal{GRS}$

Broadcast and convergecast. Informally, a wave of communication consists of a broadcast phase from the root node, $\mathcal{GRS}^r$ to all other nodes, followed by a convergecast phase from the receiving nodes to $\mathcal{GRS}^r$. As the system is asynchronous, some nodes may execute faster than others. Thus, the convergecast phase may be initiated in some parts of the network while other parts of the network are still in the broadcast phase.

$i \downarrow$ and $i \uparrow$ mean that $i$ is in the broadcast and convergecast phase, respectively.

Definition 5 (Wave) A wave (denoted as $W(r)$) is a finite minimal sequence of configurations $c_1, \ldots, c_k$ such that:

1. In $c_1$, $\mathcal{GRS}^r \downarrow \land \forall i \in \mathcal{GRS}^s \cup \mathcal{GRS}^l$, $i \uparrow$.

2. $\forall i \in \{1, \ldots, k\}$, $c_{i+1}$ is reached from $c_i$ if one or more of the following actions is executed:

   (a) $j \in \mathcal{GRS}^s$ changes from $j \uparrow$ to $j \downarrow$ if and only if $P^j \downarrow$.

   (b) $j \in \{\mathcal{GRS}^r\} \cup \mathcal{GRS}^s$ changes from $j \downarrow$ to $j \uparrow$ if and only if $\forall j' \in C^j$, $j' \uparrow$.

3. In $c_{k+1}$, $\mathcal{GRS}^r \downarrow \land \forall i \in \mathcal{GRS}^s \cup \mathcal{GRS}^l$, $i \uparrow$.

It is often desirable to chain these wave of communication.

Definition 6 (Wave Chain) A $(k, r)$-wave chain of communication is a sequence of $k$ waves of communication. It is denoted by $WC(k, r)$.

Color synchronization. In order to distinguish the successive waves, the waves are colored by adding color to processors. The color of processor $i$ will be denoted as $i_c$ which may take values in $\{\text{black, white}\}$.

Definition 7 (Colored Wave) A $b$-colored wave is a finite minimal sequence of configurations $c_1, \ldots, c_k$ such that:
1. In $c_1$, $\text{GRS}^r \downarrow \land \text{GRS}^r |_{c_1} = b \land \forall i \in \text{GRS}^s \cup \text{GRS}^l, i \uparrow \land i |_{c_1} = \text{GRS}^r |_{c_1}$.

2. $\forall i \in \{1, \ldots, k\}, c_{i+1}$ is reached from $c_i$ if one or more of the following actions is executed:

   (a) $\text{GRS}^r$ changes from $\text{GRS}^r \downarrow$ to $\text{GRS}^r \uparrow$ and $\text{GRS}^r |_{c_1}$ changes to $\neg \text{GRS}^r |_{c_1}$ if and only if $\forall j \in C^\text{GRS}^r, j |_{c_1} = \text{GRS}^r |_{c_1} \land j \uparrow$.

   (b) $j \in \text{GRS}^s$ changes from $j \uparrow$ to $j \downarrow$ and $j |_{c_1}$ changes from $b$ to $\neg b$ if and only if $P_j \downarrow$ and $P_j |_{c_1} \neq j |_{c_1}$.

   (c) $j \in \text{GRS}^l$, $j |_{c_1}$ changes from $b$ to $\neg b$ if and only if $P_j \downarrow$ and $P_j |_{c_1} \neq j |_{c_1}$.

3. In $c_{k+1}$, $\text{GRS}^r \downarrow \land \text{GRS}^r |_{c_1} = \neg b \land \forall i \in \text{GRS}^s \cup \text{GRS}^l, i \uparrow \land i |_{c_1} = \text{GRS}^r |_{c_1}$.

The color of a colored wave is denoted by $W(r) |_{c_1}$.

**Definition 8 (Color Synchronization)** A $(k, r)$-wave chain, $\text{WC}(k, r) = W(r)_1, W(r)_2, \ldots W(r)_k$ is called a color synchronized wave chain if and only if the following conditions are satisfied:

1. $\forall i \in \{1, \ldots k-1\}, W(r)_i |_{c_1} \neq W(r)_{i+1} |_{c_1}$.

2. $\forall i \in \{1, \ldots k\}, W(r)_i |_{c_1} \in \{\text{black, white}\}$.

A color synchronized wave chain is denoted by $\hat{\text{WC}}(k, r)$.

**Problem.** The *global rooted synchronization* is achieved on a tree network when every computation in the tree is a color synchronized wave chain $\hat{\text{WC}}(\infty, r)$.

### 3.2 Global Rooted Synchronization Algorithm

**Overview.** The global rooted synchronization ($\text{GRS}$) is a technique for achieving the synchronization over successive wave chains. We consider three kinds of nodes in the tree network: the *root*, the *leaf* nodes, and the others, called the *intermediate* nodes.

The leaf and intermediate nodes act as slaves to their parents (and hence to the root node) by copying their parent’s color and forwarding the parent’s message down the tree. Only the root node may initiate a new color for a new wave. Eventually, all nodes will be color synchronized with the root node, and that the root node will initiate a new color synchronized wave.

**Informal description.**

1. The *root* is considered as the leader of the system. When its color is $c$, it initiates a $c$-colored wave by sending *broadcast* messages to all its children (which are either intermediate or leaf nodes). Upon receiving $c$ colored *convergecast* messages from all its children, the root initiates a new wave with a color different from $c$. 


2. Upon receiving a message from their parent, the intermediate nodes compare the color of the parent with theirs. If the colors are different, the intermediate nodes copy the color of the parent as their own color and forwards the message of the parent by initiating a broadcast phase down the tree. This process synchronizes the intermediate nodes with their parent. Upon receiving convergecast messages from all their children, if the color of the intermediate nodes match with that of all their children, the intermediate nodes forward the convergecast message to their parent.

3. The leaf nodes act almost as the intermediate nodes except that since they don’t have any children, they simply copy their parent’s color during the broadcast phase or the parent’s color is different than theirs. This completes the broadcast phase of a colored wave. Then the leaf nodes initiate the convergecast phase by sending the message back to their parent.

**Formal description.** The GRS algorithm uses the following two binary variables at each processor.

1. $c$: The color of the node. It may take the values black or white, and may be negated (assuming $\neg$ white = black and $\neg$ black = white).

2. $t$: The phase of the wave communication. It may take values $t_b$ (the broadcast phase) or $t_c$ (the convergecast phase).

We assume that an underlying spanning tree protocol maintains, at each node $i$, the set of its children (denoted by $C^i$) and its parent (denoted by $P^i$). The GRS scheme is presented in Algorithm 3.1 where each variable $v$ of node $i \in \{P\} \cup C$, is referenced as $i_v$ and the variable $v$ of the node being described is referenced by $v$.

### 4 Applications

In this section, we provide several applications for our global synchronizer algorithm (Algorithm 3.1). These applications include the global clock maintenance algorithm (in Section 4.1) as well as global calculus algorithms (in Section 4.2).

#### 4.1 Clock maintenance

We show how to use Algorithm 3.1 as a basis for building a digital clock synchronization algorithm. Although being built on top of a general scheme, this algorithm improves the best known memory complexity to-date while maintaining asymptotically optimal time complexity.

**Informal Problem Specification.** Nodes maintain a clock variable, which may represent the program counter of some external nonterminating algorithm running a sequence of $k$ actions:

$$a_0, a_1, \ldots, a_{k-1}, a_0, a_1, \ldots$$

Then all nodes are required to perform the same external algorithm action “simultaneously”. In fact, the clocks at all nodes must read the same when performing the external algorithm actions, and increment in unison.
Algorithm 3.1 The GRS Mechanism.

Field Variables:
- $c$: The color of the node
- $t$: The phase of the wave

Local Variables:
- $C$: The set of children of the node
- $P$: The parent of the node

Constants:
- $t_b$: The broadcast phase (a possible value of $t$)
- $t_c$: The convergecast phase (a possible value of $t$)

\begin{align*}
\text{\{This rule is executed by the root node only\}}
R_1: \quad & (\forall k \in C, k_c = c \land k_t = t_c) \implies
\quad c \leftarrow \neg c \; ; \; t \leftarrow t_b \\
\text{\{This rule is executed by the intermediate nodes only\}}
R_2: \quad & (P_c \neq c) \implies
\quad c \leftarrow P_c \; ; \; t \leftarrow t_b \\
\text{\{This rule is executed by the intermediate nodes only\}}
R_3: \quad & (\forall k \in C, k_c = c \land k_t = t_c) \implies
\quad t \leftarrow t_c \\
\text{\{This rule is executed by the leaf nodes only\}}
R_4: \quad & (P_c \neq c \lor t = t_b) \implies
\quad c \leftarrow P_c \; ; \; t \leftarrow t_c
\end{align*}

Informal Solution. We add a clock variable to every node in the tree. Algorithm 3.1 is slightly modified in the following manner:

- The intermediate and leaf nodes take the clock value of their parent while changing their color value.
- The root node increments its clock variable modulo $k$ (where $k$ is the number of steps of the external algorithm) while changing its color value.

From Proposition 2, within $O(h)$ time, the system will reach a color synchronized configuration—the clock variables of all nodes will have the same value. Starting from this configuration, whenever the root will initiate the next colored wave (with a different color), it will increment its clock variable. Thus, after $O(h)$ time, the clocks will be synchronized. This stabilization time was shown to be asymptotically optimal in [6].

The space complexity of this modified global synchronization algorithm is still $O(1)$ (or, $O(\delta)$ if the memory required to maintain the tree topology is taken into account), since the color variable uses $O(\log k)$ memory and $k$ is fixed for a tree network of any size. This is an improvement over other asynchronous clock synchronization or unison algorithms such as the algorithm in [6] (which requires $O(\log n \times \log D)$), the algorithm in [3] (which uses $O(\log n)$), and the algorithm in [11] (which needs $O(\log n^2)$. The synchronous clock synchronization
algorithms such as [9], [13], and [15], require at least $O(1)$ memory and $O(h)$ stabilization time. So, our clock synchronizer algorithm competes well with the synchronous algorithms as well.

### 4.2 Global Calculus

**Informal Problem Specification.** Nodes are required to perform a global calculus on variables which are distributed on a network. We require the operation to be performed be *associative* and *commutative*. Such operations include computing the *maximum*, *minimum*, *adding* variables all all nodes, etc.. After the system stabilizes, nodes are required to contain the correct value of the result (the maximum, minimum, sum, etc.).

**Informal Solution.** Let $f$ be the commutative and associative operation to be performed. We use the following scheme to compute $f$:

- *Convergecast* phases are used to compute the $f$ operation.
- *Broadcast* phases are used to broadcast the result of the previous $f$ operation to all nodes in the network.

We add three variables at each node: *value*, which contains the value on which the operation $f$ is to be performed, *lresult* which contains the result (called the *local result*) of the $f$ operation computed over the nodes in the subtree rooted at the node, and *gresult*, which holds the result (called the *global result*) of the $f$ operation.

Algorithm 3.1 is slightly modified in the following manner to compute the global calculus:

- *$f$ Calculus*: The leaf nodes perform $f$ on their own *value*, and write the result in their *lresult* variable, and set their phase type to *convergecast*. The intermediate and root nodes perform $f$ on the *lresult* variables of the nodes in the subtrees rooted at them, write the result in their own *lresult* variable, and set their phase type to *convergecast*.

- *Result Broadcasting*: The root copies its *lresult* value to its *gresult* variable while changing its color. The leaf and intermediate nodes copy their parent’s *gresult* value to their *gresult* variable while changing their color.

Not only our algorithm is very simple, its time and space requirements are also very small—$O(h)$ time and $O(v)$ space, where $v$ is the size of the *value* variable.

### 5 Proof of Correctness

We first show that within finite time, all nodes (other than the root) will color synchronize with their parent.

**Lemma 1 (Parent Synchronization)** $\forall c_1 \in C$, $\forall e \in E_{c_1}$, $\forall j \in \text{GRS}^* \cup \text{GRS}^l$, $\exists i \geq 1$, $j_c = P_{c_1}^i$ in $c_1$. 

9
Proof: Assume that there exists some node $j \in \mathcal{GRS}^d$ such that $j_c \neq P^d_i$. Then by rule $R_2$, $j_c = P^d_i$. Similarly, assume that there exists some node $j \in \mathcal{GRS}^l$ such that $j_c \neq P^l_i$. Then by rule $R_4$, $j_c = P^l_i$. Thus, no intermediate or leaf node can have a color different from its parent forever.

Then we demonstrate the liveness property of the $\mathcal{GRS}$ algorithm by proving that the color of the root node cannot remain the same for an infinite amount of time.

Lemma 2 (Liveness) \( \forall c_1 \in \mathcal{C}, \forall e \in \mathcal{E}_{c_1}, \mathcal{GRS}^t_e \) changes infinitely often.

Proof: Assume the contradictory, i.e., assume that \( \exists c_1 \in \mathcal{C}, \exists e \in \mathcal{E}_{c_1}, \mathcal{GRS}^t_e \) never changes (i.e., Rule $R_0$ is never enabled). This means that \( \exists n \in C^{\mathcal{GRS}^d}, n_t = t_b \) or \( n_c \neq \mathcal{GRS}^t_e \) forever. By Lemma 1, the situation when \( n_c \neq \mathcal{GRS}^t_e \) cannot occur. The condition \( n_t = t_b \) means \( \exists n' \in C^n, n'_t = t_b \), or \( n'_c \neq n_c \) forever. By induction on the height of the tree, we get \( \exists t \in \mathcal{GRS}^l, l_t = t_b \) or \( l_c \neq P^l_i \) forever. Now, Rule $R_4$ is enabled, which when executed will set \( l_t = t_c \) and \( l_c = P^l_i \). We arrived at the contradiction.

Now we define a predicate on configurations that characterizes the desirable properties for both the correctness and convergence proofs.

Definition 9 (Synchronization Predicate) We define the synchronization predicate, $\mathcal{L}_{\mathcal{GRS}}$ on $\mathcal{C}$ such that $\mathcal{L}_{\mathcal{GRS}} = L_{\mathcal{GRS}}^h$ where $h$ is the height of the tree and

\[
L_{\mathcal{GRS}}^h = \left\{ \begin{array}{l}
\forall n \in V^d, n_c = \mathcal{GRS}^t_e \\
\land \\
\mathcal{GRS}^t_e = t_b \\
\land \\
\forall n' \in (\mathcal{GRS}^s \cup \mathcal{GRS}^l) \cap V^d, n'_t = t_c 
\end{array} \right\}
\]

Informally, the configurations of $\mathcal{C}$ that match $L_{\mathcal{GRS}}^h$ are those where all nodes up to distance $d$ have the same color and all nodes up to distance $d$ but root have their phase set to convergecast while the root has its phase equal to broadcast.

We will show that from any configuration that satisfies $\mathcal{L}_{\mathcal{GRS}}$, any computation of the $\mathcal{GRS}$ system satisfies the specification of the $\mathcal{GRS}$ problem, namely $WC(\infty, r)$.

Lemma 3 (Correctness) \( \forall c_1 \in \mathcal{L}_{\mathcal{GRS}}, \forall e \in \mathcal{E}_{c_1}, e = WC(\infty, r) \) where $h$ is the height of the tree.

Proof: From Definition 6, it follows that $e$ starts with $WC(2, r)$. Similarly, we can also claim that $e$ starts with $W(r)$.

For any node $i$, we replace $i \downarrow$ to $i_t = t_b, i \uparrow$ to $i_t = t_c$, and $i \mid c$ to $i_c$. Then any $l \in L_{\mathcal{GRS}}$ is a valid initial configuration for $W(r)$. All nodes have the same color in $l$. We call this color $c(l)$. Moreover, any rule that makes the processors change their state satisfies the conditions given in Definition 7. Thus, any computation step is also acceptable.

To establish the rest of the proof, we need to show that within a finite number of steps, we will reach another configuration $l' \in L_{\mathcal{GRS}}$ such that $(\forall n \in V, n_c = \neg c(l) \land \mathcal{GRS}^t_e = t_b) \land (\forall n' \in \mathcal{GRS}^s \cup \mathcal{GRS}^l, n'_t = t_c)$. The only enabled rule in $l$ is $R_1$ which when executed changes the color of $\mathcal{GRS}^r$ to $\neg c(l)$. From now on, $R_1$ is not enabled until all nodes in $C^{\mathcal{GRS}^r}$ have color
$\neg c(l)$ and are in convergecast phase. The children of the root cannot change their phase to convergecast until they receive an acknowledgment from all their children. From Rules $\mathcal{R}_2$ and $\mathcal{R}_4$, nodes in $\mathcal{GRS}^\ast \cup \mathcal{GRS}^1$ take their parents’ new color $\neg c(l)$, and the convergecast can be initiated only by the leaf nodes that have taken the new color. When all nodes have the same new color $\neg c(l)$, only $\mathcal{R}_2$ can be applied, thus continuing the convergecast phase towards the root. After a finite time, all nodes have the same color $\neg c(l)$, and the root is in the broadcast phase, while all other nodes are in the convergecast phase. This last configuration satisfies $L_{\mathcal{GRS}}$.

We now prove that for any initial configuration and any possible computation of the $\mathcal{GRS}$ algorithm, we reach a configuration that satisfies $L_{\mathcal{GRS}}$.

**Lemma 4 (Convergence)** $\forall c_\alpha \in \mathcal{C}, c_\alpha \triangleq \text{true}, \forall e \in E_{c_\alpha}, (e = c_1, c_2, \ldots) \land c_\alpha = c_1, \exists i \geq 1, c_i \triangleq L_{\mathcal{GRS}}^k$.

**Proof:** We will prove the following property, $P(k)$ by induction on the height of the tree:

$$P(k) \equiv \forall c_\alpha \in \mathcal{C}, c_\alpha \triangleq \text{true}, \forall e \in E_{c_\alpha}, (e = c_1, c_2, \ldots) \land c_\alpha = c_1, \exists i \geq 1, c_i \triangleq L_{\mathcal{GRS}}^k$$

**Base Case.** We show that $P(1)$ is satisfied. We consider only the root node $\mathcal{GRS}^r$ and its children. We need to consider two situations depending on there exists a child with a color different from the root or not:

1. $\exists n \in \mathcal{GRS}^r, n_c \neq \mathcal{GRS}^r$. Since it has at least one child with a different color from itself, the root cannot change its color ($\mathcal{R}_1$ is not enabled). But, by Lemma 1, $n$ will change its color.

2. $\forall n \in \mathcal{GRS}^r, n_c = \mathcal{GRS}^r$. First, since all children have the same color as their parent, they cannot change their current color (Rule $\mathcal{R}_2$ is not enabled and Rule $\mathcal{R}_4$ does not change the color). Second, a child cannot change its phase if it is already set to convergecast (the only way to do this is by executing $\mathcal{R}_3$, which is not enabled).

Now we need to consider two situations depending on the phase value of the children:

(a) $\exists n \in \mathcal{GRS}^r, n_t = t_b$. This kind of children cannot be in the broadcast phase forever. Otherwise, it implies that $\exists n' \in C^n, n'_t = t_b$ or $n'_c \neq n_c$ forever. By induction on the height of the tree, this implies that $\exists i \in \mathcal{GRS}^l, l_t = t_b$ or $l_c \neq P_c^l$ forever. Then Rule $\mathcal{R}_4$ is enabled, which when executed will falsify the above condition.

(b) $\forall n \in \mathcal{GRS}^r, n_t = t_c$. In this case, $P(1)$ is satisfied.

**Induction Step.** Assume that there exists some $k \ (1 \leq k \leq h - 1)$ such that $P(k)$ is satisfied. We will show that $P(k + 1)$ is also satisfied.

Since $P(k)$ is satisfied, starting from any initial configuration, and for any computation of $\mathcal{GRS}$, there exists a configuration $c_i$ such that all nodes up to distance $n$ from the root have the same color and also all these nodes (except the root) have their phase variable equal to convergecast. In $c_i$, the only rule enabled at any node up to distance $k$ is $\mathcal{R}_1$ (which is enabled at the root node). By executing $\mathcal{R}_1$, the root changes its color. By Lemma 1 and by induction on $k$, all nodes up to distance $k$ take the new color.

Now, consider the nodes that are at distance exactly $k$ from the root and that are not the leaf nodes (this set must be non-empty since the height of the tree is at least $k + 1$). We will
refer to these nodes as $V^k$. These nodes already have the same color as the root and have their phase variable set to broadcast (they executed Rule $R_2$).

By Lemma 1, the children of nodes in $V^k$ can only take their parent’s color and cannot change it until their parents change their color. Following the similar reasoning, we can say that the children of the root cannot change their color until the root receives the convergecast information from its children. Following the same reasoning as in the base case, the children of the nodes in $V^k$ must change their phase to convergecast and stay in this phase. Eventually, the convergecast phase continues by executing Rule $R_3$. Since all children of nodes in $V^k$ have the same color and have the phase set to convergecast, $P(k+1)$ is satisfied.

We saw that $P(1)$ is true, and if $P(k)$ is true for $1 \leq k \leq h-1$, then $P(k+1)$ is also true. Consequently, $P(h)$ is verified, and the lemma is proven.

Finally we show that the GRS system is self-stabilizing for the color synchronization problem.

**Theorem 1** $\forall c_\alpha \in \mathcal{C}, c_\alpha \triangleright \text{true}, \forall e \in \mathcal{E}_{c_\alpha}, (e = c_1, c_2, \ldots) \land c_\alpha = c_1, \exists i \geq 1, \forall e' \in \mathcal{E}_{c_i}, e' = \widehat{WC}(\infty, r)$.

**Proof:** The proof is a direct consequence of lemmas 3 and 4.

### 6 Complexity

In this section, we analyze both the space (Definition 3) and time complexity (Definition 4) for the GRS algorithm (Algorithm 3.1).

**Proposition 1 (Space Complexity)** The memory requirement at each processor for the GRS algorithm is $O(1)$.

**Proof:** Algorithm 3.1 uses only two variables at each processor (other than the memory required to maintain the tree) — $c$ and $t$. The value of $c \in \{\text{black}, \text{white}\}$. The value of $t \in \{t_b, t_c\}$. Thus, in total, 2 bits of memory at each processor is sufficient.

**Note 1** If we take into account the memory space needed to maintain the tree topology, then this space complexity of the GRS algorithm is $O(\delta)$, where $\delta$ is degree of the network.

**Proposition 2 (Time Complexity)** The time complexity, $T_h$ for the GRS algorithm is $O(h)$, where $h$ is the height of the tree.

**Proof:** Assume that $V^d$ denotes the set of nodes that are at a distance of exactly $d$ from the root. Let $r|_c^i$ be the color of the root at time $T_i$.

Let $T_0$ be the time when all faults cease to occur. We will denote the time, $i$ time units starting from $T_0$, by $T_i$. At $T_1$, all nodes in $V^1$ have color $r|_c^0$. Then at each subsequent time unit, $T_i$ ($2 \leq i \leq h$), all nodes in $V^i$ have color $r|_c^0$. Between $T_0$ and $T_h$, two situations may arise, depending on whether the root has changed its color or not between the time $T_0$ and $T_h$:

1. The root has not changed its color. Then at time $T_h$, all nodes in $V^h$ have color $r|_c^0$. Since $h$ is the height of the tree, all nodes have the same color at $T_h$. Thus, by the time $T_{2h}$, all nodes in $\mathcal{GRS}^* \cup \mathcal{GRS}^L$ will be in the converge-cast phase while the root, $\mathcal{GRS}^r$ will be in the broadcast phase, and all nodes have the color $r|_c^0$. 


2. The root has changed its color at $T_i$ ($1 \leq i \leq h$) to color $\neg r\mid c^0$. Till $T_1$, all nodes in $\overline{V}^{i-1}$ had color $r\mid c^0$. Similarly, nodes in $\overline{V}^{i-2}$ had color $r\mid c^0$ till $T_2$, and so on. As there is at least a one time unit gap between the $r\mid c^0$ and $\neg r\mid c^0$ waves, the $\neg r\mid c^0$ wave reaches only the $r\mid c^0$ colored nodes. Then, it is impossible that any node $j$, reached by the $\neg r\mid c^0$ wave, is in the converge-cast phase unless: (i) $j$ is a leaf or (ii) $j$ is an intermediate node whose subtree is completely colored with the color $\neg r\mid c^0$ and is in the converge-cast phase. Thus, by time $T_{i+2}$, all nodes have the same color $\neg r\mid c^0$, and by time $T_{i+2h}$, all nodes in $GRS^s \cup GRS^l$ will be in the converge-cast phase while the root, $GRS^r$ will be in the broadcast phase, and all nodes have the color $\neg r\mid c^0$.

So, in at most $i + 2 \times h \leq 3 \times h$ time units, the tree reaches a configuration that satisfies the synchronization predicate, $L_{GRS}$ (Definition 9).

The space and time complexity should be compared with those of [19] which also provides a global synchronization scheme for tree networks. The stabilization time of the algorithm in [19] is $O(h)$, which is asymptotically the same as ours. But, we improve the space complexity—it is $O(\log n)$ in [19] whereas it is $O(1)$ for our scheme.

7 Conclusions

We presented a self-stabilizing scheme to achieve global synchronization in a tree network. Our technique, called the global rooted synchronization, can be used to build a compiler that converts the non-stabilizing synchronous algorithms into their self-stabilizing asynchronous counterpart. Our technique improves the space complexity over the previous approaches to the global synchronization while maintaining the asymptotically optimal stabilization time.

References


