Self-Stabilizing Neighborhood Synchronizer in Tree Networks

Colette Johnen[1] Luc O. Alima[2] Ajoy K. Datta[3*]
Sébastien Tixeuil[1]

[1] Laboratoire de Recherche en Informatique, Université de Paris-Sud, France
[2] Unité d'Informatique, Université catholique de Louvain, Belgium
[3] Dept of Computer Science, University of Nevada Las Vegas

Abstract

We propose a self-stabilizing synchronization technique, called the Neighborhood Synchronizer *($\mathcal{NS}$), that synchronizes nodes with their neighbors in a tree network. The $\mathcal{NS}$ scheme has extremely small memory requirement—only 1 bit per processor. Algorithm $\mathcal{NS}$ is inherently self-stabilizing. We apply our synchronizer to design a broadcasting algorithm $\mathcal{BA}$ in tree networks. Algorithm $\mathcal{BA}$ is also inherently self-stabilizing and needs only $2h + 2m - 1$ rounds to broadcast m messages, where h is the height of the tree.*

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 [11] 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 [17], [18], and [19]. Informally, a synchronizer is a protocol than allows asynchronous systems to simulate the behavior of synchronous systems.

Related Work. The research in the area of synchronizers started from the seminal work of Awerbuch [4]. This work is not self-stabilizing.

*Supported in part by a sabbatical leave grant from the University of Nevada Las Vegas.

Any self-stabilizing reset protocol [3, 1, 6] can be combined with the protocol of [4] to design a self-stabilizing synchronizer. Some general self-stabilizing synchronizers have been proposed in [7, 5, 20].

A global self-stabilizing synchronizer for tree networks is proposed in [2]. The space complexity of the algorithm is 2 bits and the time complexity is $O(h)$ rounds. The notion of the legitimate state as defined in [11] is modified in [16] to allow concurrent moves in different branches of a tree. Although Kruijer's paper did not discuss the problem of synchronization, the algorithm he presented can also be considered as a global synchronizer in tree networks.

A space optimal self-stabilizing propagation of information with feedback (PIF) scheme in tree networks is presented in [8]. The stabilizing time of this PIF scheme is $h - 1$ steps and the space requirement is 3 states.

A local and a global synchronizer are presented in [13]. The local synchronizer is designed to synchronize colors of a two processor system and is also used to design the global synchronizer on a tree network. The stabilization time of the global synchronizer is $O(\Delta h)$ where Δ is the degree of the tree and h is the height of the tree.

Our Contribution. We propose a self-stabilizing synchronization technique, called the *Neighborhood Synchronizer* ($\mathcal{NS}$), that synchronizes nodes with their neighbors in a tree network. The $\mathcal{NS}$ scheme has a constant stabilization time and requires only 1 bit of memory per processor.

We then present a solution to the broadcasting problem in tree networks as an application of the neighborhood synchronizer. The broadcasting algorithm needs only $2h + 2m - 1$ rounds to broadcast m messages. The local synchronizer of [13] synchronizes only *two* processors, whereas Algorithm $\mathcal{NS}$ presented in this paper synchronizes a processor with all its neighbors (parent and children in the tree network).

The self-stabilizing spanning tree construction algorithms have been proposed in [1], [3], [5], [10], [12], and [15]. Any of these algorithms can be combined with

our synchronizer to design a synchronizer for a general network. In [14], Gouda and Haddix proposed a self-stabilizing alternator on a chain that transforms any linear system that is stabilizing, but works under a central daemon, to one that is stabilizing and works under a distributed daemon. The Neighborhood Synchronizer is a self-stabilizing alternator on the tree networks as defined in [14].

Outline of the Paper. In Section 2, we describe the distributed systems and the model we consider in this paper. The synchronization scheme, called *neighborhood synchronizer* and its correctness proof are presented in Section 3. We present a self-stabilizing broadcast algorithm as an application of the local synchronizer in Section 4. The correctness of the broadcast algorithm is given in Sections 4.2. Finally, we give the concluding remarks in Section 5.

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.

System. A *distributed system* is an undirected connected graph, $D = (V, E)$, where V is the set of nodes ($|V| = n$) and E is the set of edges. Nodes represent *processors* and edges represent *bidirectional communication links*. We consider networks which are *asynchronous* and *tree structured*. We denote the *root* processor by r, the set of *leaf* processors by L, and the set of *internal* processors by I. So, the set of all processors, $V = \{r\} \cup I \cup L$. We denote the processors by $p :: p \in \{1..n\}$ and the root processor by r. The numbers, $1..n$, are used for notation only, since no processor, except the root, is aware of its identity. A communication link (p, q) exists iff p and q are neighbors. Each processor p maintains its set of neighbors, denoted as N_p. The *degree* of p is the number of neighbors of p, i.e., equal to $|N_p|$. We assume that each processor p ($p \neq r$) knows its parent, denoted by P_p. We consider N_p and P_p as constants in our algorithm. An actual implementation will maintain them by using a self-stabilizing underlying protocol. The height of a tree is denoted by h. We use h_p to denote the height of the subtree rooted at p.

Programs. 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 and is of the following form:

$$< label >:: < guard > \longrightarrow < statement >$$

The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. 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 p is called a *step* of p. Since the system is asynchronous, we define a time unit, *round*, to consider the slowest processor of the system. We define a *round* as a computation step during which all processors which have at least one guard set to true, execute an action. 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 sequel, we refer to the state of a processor and system as a (*local*) *state* and *configuration*, respectively.

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. During a computation step, one or more processors execute an action and a processor may take at most one action. This execution model is known as the *distributed daemon* [9].

We use the notation $Enable\,(A, p, \gamma)$ to indicate that the guard of the action A is true at processor p in the configuration γ. We write $Enable(p, \gamma)$ if the guard of at least one action is true at p in γ. We assume a *weakly fair* daemon, meaning that if processor p is continuously $enabled$, p will be eventually chosen by the daemon to execute an action. The set of computations of a protocol $\mathcal{P}$ in system S starting with 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 $\mathcal{E}$. A configuration β is *reachable* from α, denoted as $\alpha \rightsquigarrow \beta$, if there exists a computation $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots) \in \mathcal{E}_\alpha (\alpha = \gamma_0)$ such that $\beta = \gamma_i$ ($i \geq 0$).

Predicates. 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* $x \in \mathcal{X}$, $x \vdash \mathsf{true}$.

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

Definition 2.1 (Closed Attractor) *Let X and Y be two predicates defined on $\mathcal{C}$ of system $\mathcal{S}$. Y is a closed attractor for X if and only if the following conditions are true:*

(i) $\forall \alpha \vdash X : \forall e \in \mathcal{E}_\alpha : e = (\gamma_0, \gamma_1, \ldots) :: \exists i \geq 0, \gamma_i \vdash Y$

(ii) $\forall \alpha \vdash Y : \forall e \in \mathcal{E}_\alpha : e = (\gamma_0, \gamma_1, \ldots) :: \forall i \geq 0, \gamma_i \vdash Y$.

We denote this relation as $X \triangleright Y$.

Informally, $X \triangleright Y$ means that in any computation $\in \mathcal{E}_\alpha$, starting from an arbitrary configuration satisfying X, the system is guaranteed to reach a configuration which satisfies Y, and also, Y is closed.

Definition 2.2 (Self-stabilization) *A protocol $\mathcal{P}$ is self-stabilizing for a 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:*

(i) $\forall \alpha \vdash \mathcal{L}_\mathcal{P} : \forall e \in \mathcal{E}_\alpha :: e \vdash \mathcal{SP}_\mathcal{P}$ *(correctness).*

(ii) $\mathsf{true} \triangleright \mathcal{L}_\mathcal{P}$ *(closure and convergence).*

3. Neighborhood Synchronizer ($\mathcal{NS}$)

In this section, we provide a scheme to implement the synchronization among the neighboring processors. We first give the specification of the $\mathcal{NS}$ problem. Then we describe the scheme informally, followed by Algorithm $\mathcal{NS}$. Finally, we prove the correctness of Algorithm $\mathcal{NS}$.

Problem to be Solved. We are now ready to define the specification, $\mathcal{SP}_{\mathcal{NS}}$ of Algorithm $\mathcal{NS}$. We consider a computation e to satisfy $\mathcal{SP}_{\mathcal{NS}}$ if between every two successive actions executed by a processor in e, all its neighbors execute exactly one action. We also require Algorithm $\mathcal{NS}$ to be self-stabilizing.

3.1. Informal Description

Every processor uses a binary $color$ variable, c to indicate its change of state to its neighbors. The main idea about the neighborhood synchronization is as follows: An internal processor p changes its state (the $color$) variable, c, only when it finds that all its children have the same value as c and its parent has a new (different) value of c. The protocol of the root and leaf processors are similar except the fact that the root has no parent and the leaves do not have any children.

The Neighborhood synchronization can be applied when there is a need for simulating a reliable message passing mechanism using a register-based communication model. The basic scheme for this application is as follows: The root sends a new message to its children and then waits until all its children read that new message. When that happens, the root can send another message. An internal processor reads a new message from its parent only when it finds that all its children have read its previous message. The leaves simply read a new message from their parent whenever the parent sends a new message. The root changes c to signal to its children that it has sent a new message. The internal and leaf processors change their value of c to acknowledge to their parent that they have read the previous message sent by the parent, and also to indicate to their children that they now hold a new message.

3.2. Algorithm $\mathcal{NS}$

The Neighborhood Synchronizer algorithm $\mathcal{NS}$ is shown in Algorithm 3.1. Every processor i maintains a variable c_i, the state of i. We denote the set of children of i by Cld_i, where $Cld_i = N_i \setminus \{P_i\}$.

Algorithm 3.1 ($\mathcal{NS}$) Neighborhood Synchronizer Algorithm for processor i.

Variable:
c_i: The $color$ variable.
Constants:
Cld_i: The set of children.
P_i: The parent processor.
Actions:
{**For the root**}
$S_1 :: \forall j \in Cld_i :: c_j = c_i \longrightarrow c_i := \neg c_i$

{**For the internal processors**}
$S_2 :: c_{P_i} \neq c_i \wedge (\forall j \in Cld_i :: c_j = c_i) \longrightarrow c_i := c_{P_i}$

{**For the leaf processors**}
$S_3 :: c_{P_i} \neq c_i \longrightarrow c_i := c_{P_i}$

3.3. Correctness of Algorithm $\mathcal{NS}$

We will first prove the liveness of the algorithm. Then, the proof of the desired behavior of the computation of Algorithm $\mathcal{NS}$ follows from the liveness result. The following property follows directly from Algorithm 3.1.

Property 3.1 $\forall \gamma \in \mathcal{C} : \forall i \in V :: Enable(i, \gamma) \Rightarrow (\forall j \in N_i :: \neg Enable(j, \gamma))$.

Lemma 3.1 *For every processor $i \in (\{r\} \cup I)$, the following is true: $\forall \gamma \in \mathcal{C} : \exists \gamma\prime : \gamma \rightsquigarrow \gamma\prime : \forall j \in Cld_i :: c_j = c_i$ in $\gamma\prime$.*

Proof: Assume that i executes an action (S_1 or S_2) during a computation e. Then, all the children of i must have the color of i before i executes the action (see the guard of S_1 and S_2).

Let us now consider the case where i never executes an action during the computation e (i.e., i never changes its

color). Once a child j of i has the color of i, j cannot execute any action. We will prove this case by induction on the height of the subtree rooted at i.

(i) **Base Case:** $h_i = 1$, i.e., i is a parent of some leaf processors. Assume that there exists a processor $j \in Cld_i$ such that c_j never becomes equal to c_i during a computation e. Then, in all configurations in e, S_3 remains enabled until j executes S_3. By fairness, j will eventually execute S_3 and $c_j = c_i$ becomes true. Eventually, all the children of i will get the color of i.

(ii) **Hypothesis:** Assume that the lemma is true for $0 < h_i \leq m, m \leq h-1$. Assume that there exist two processors i and j such that $h_i = m+1, j \in Cld_i$, and j never gets the $color$ of i during e. j cannot execute S_2 because that would make $c_j = c_i$. By the induction hypothesis, the system will reach a configuration γ where the children of j will get the $color$ of j. Then $Enable(S_2, j, \gamma)$ will be true and will remain true until j executes S_2 (by Property 3.1). By fairness, j will eventually execute S_2 and $c_j = c_i$ becomes true. Once $c_j = c_i$, j cannot change its color. Thus eventually, all the children of i will get the color of i.

□

Now, we are ready to prove the liveness of Algorithm $\mathcal{NS}$.

Lemma 3.2 (Liveness) *$\forall e \in \mathcal{E}, \forall i \in V$, i executes an action infinitely often.*

Proof: We will prove this by contradiction. Assume that there exists at least one processor that stops executing any action from the configuration γ during a computation e. Let i be the processor nearest to the root among these processors. By Lemma 3.1, in some configuration $\gamma\prime$, $\gamma \rightsquigarrow \gamma\prime$, all children of i will have the same $color$ as i. As i cannot change its $color$, no child of i can also change its $color$ in any configuration from $\gamma\prime$ onwards in e.

(i) Assume that $i = r$. Then $Enable(S_1, i, \gamma\prime)$ is true. By fairness, i will eventually execute S_1.

(ii) Assume that $i \neq r$ and $c_i \neq c_{P_i}$. Then either $Enable(S_2, i, \gamma\prime)$ (if $i \in I$) or $Enable(S_3, i, \gamma\prime)$ (if $i \in L$) will be true. By fairness, i will eventually execute S_2 or S_3.

(iii) Assume that $i \neq r$ and $c_i = c_{P_i}$. P_i will eventually execute an action and change its $color$ because according to the hypothesis, all ancestors of i infinitely change their $color$. After P_i executes its action, $c_i \neq c_{P_i}$ will be true. Now, P_i cannot change its $color$ again since i does not change its color. Thus, by fairness, i will eventually execute S_2 or S_3. □

Lemma 3.3 (Synchronization) *Let S_i denote an action executed by processor i. $\forall e \in \mathcal{E}, \forall i \in (\{r\} \cup I), \forall j \in Cld_i$, the projection of e on the actions of i and j can be represented by the following expression: $(S_i S_j)^\omega \cup (S_j S_i)^\omega$.*

This expression reads as :

(i) between any two actions of i (resp. j) and j (resp. i), exactly one action is performed, and

(ii) i and j execute actions infinitely often.

Proof: Follows from Lemmas 3.2 and 3.1. □

Theorem 3.1 (Self-Stabilizing) *Algorithm $\mathcal{NS}$ is a self-stabilizing neighborhood synchronizer algorithm.*

Proof: Lemma 3.3 proves that between any two actions executed by a processor, all of its neighbors execute exactly one action. This means that any computation of the $\mathcal{NS}$ Algorithm is correct. Since any initial configuration is adequate for Lemma 3.3 to be correct, the closure and convergence properties are also trivially satisfied, meaning that Algorithm $\mathcal{NS}$ is self-stabilizing. □

Complexity. Algorithm $\mathcal{NS}$ uses only one boolean variable c. Constants Cld and P are not taken into account since they are either stored in the ROM code of each processor if the network is actually a tree or maintained by an underlying tree construction algorithm. Thus, Algorithm $\mathcal{NS}$ requires only 1 bit space. Since any computation starting from any initial configuration satisfies the specification, the $\mathcal{NS}$ Algorithm has a constant stabilization time.

4. Broadcasting Algorithm ($\mathcal{BA}$): An Application of $\mathcal{NS}$

In this section, we use Algorithm $\mathcal{NS}$ as a building block to design a self-stabilizing broadcasting algorithm on a tree network. Informally, the problem to be solved is as follows: The root has an infinite sequence of messages to be broadcast to all processors of the tree. The root only waits for its children to acknowledge the receipt of the message before the root sends another message. The root does not need to wait until the previous message (sent by the root) has reached all processors of the tree. Thus, at any time, several messages may propagate down the tree, effectively, implementing a pipelining mechanism. We will show in Section 4.2 the efficiency of Algorithm $\mathcal{BA}$ due to this concurrent propagation of messages.

First, we define the problem. Then we explain informally how by making a few simple changes in Algorithm $\mathcal{NS}$, we obtain Algorithm $\mathcal{BA}$. We then show that Algorithm $\mathcal{BA}$ is self-stabilizing.

Problem Specification. We denote the distance of a processor p from the root r by δ_p. We consider a computation e to be satisfying the specification $\mathcal{SP}_{\mathcal{BA}}$ of the Broadcast Task if the following conditions are true:

(i) Every message sent by the root is eventually received by all processors in the tree in the order it was sent. We refer to this property as *Correct Delivery*.

(ii) All messages, except (possibly) the first δ_i messages received by i, were sent by the root. We refer to this property as *Message Validity*.

(iii) Algorithm $\mathcal{BA}$ is self-stabilizing.

4.1. Informal Description

We make a few simple modifications in Algorithm $\mathcal{NS}$ to design Algorithm $\mathcal{BA}$. At node i, we use an extra variable m_i to hold the current message received from the parent. The root r reads a new message and writes it in its variable m_r. The internal processors and leaf processors copy their parent's message from m_{P_i} into their own message variable, m_i.

The Broadcasting Algorithm $\mathcal{BA}$ is shown in Algorithm 4.1.

Algorithm 4.1 ($\mathcal{BA}$) Broadcasting Algorithm for processor i.

Variable:
c_i: The $color$ variable.
m_i: The message variable.
Constants:
Cld_i: The set of children.
P_i: The parent processor.
Actions:
{**For the root**}
$B_1 :: \forall j \in Cld_i :: c_j = c_i \longrightarrow$
$m_i :=$<next message>; $c_i := \neg c_i$

{**For the internal processors**}
$B_2 :: c_{P_i} \neq c_i \wedge (\forall j \in Cld_i :: c_j = c_i) \longrightarrow$
$m_i := m_{P_i}$; $c_i := c_{P_i}$

{**For the leaf processors**}
$B_3 :: c_{P_i} \neq c_i \longrightarrow m_i := m_{P_i}$; $c_i := c_{P_i}$

4.2. Correctness of Algorithm $\mathcal{BA}$

Lemma 4.1 *$\forall i \in (\{r\} \cup I), \forall j \in Cld_i$, the messages sent by i are eventually received by j in the order they were sent, with no loss or duplication.*

Proof: By Lemma 3.3 and Algorithm 4.1, after i receives a message, it cannot execute its action until all its children execute (and read the message from m_i) their action. □

As a corollary, we can state that:

Corollary 4.1 *$\forall i \in (I \cup L)$, all messages, except (possibly) the first one, received by i were sent by P_i.*

Note that the first received message by i may not have been sent or received by P_i because the message may be corrupted due to some transient faults.

Lemma 4.2 (Correct Delivery) *Every message sent by the root is eventually received by all processors in the tree in the same order it was sent.*

Proof: The proof follows from Lemma 4.1 and by using induction on the height of the tree. □

Lemma 4.3 (Message Validity) *All messages, except (possibly) the first δ_i messages received by i, were sent by the root.*

Proof: The proof follows from Corollary 4.1 and by using induction on δ_i, the distance of i from the root. □

Theorem 4.1 *Algorithm $\mathcal{BA}$ is self-stabilizing.*

Proof: From Lemmas 4.2 and 4.3, starting form any arbitrary configuration, any subsequent computation satisfies both the Correct Delivery and Message Validity predicates. So, Algorithm $\mathcal{BA}$ is correct. Since all computations satisfy the predicate $\mathcal{SP}_{\mathcal{BA}}$, Algorithm $\mathcal{BA}$ trivially satisfies the closure and convergence properties, and hence, is self-stabilizing. □

4.3. Complexity

In this section, we present the time and space requirements of Algorithm $\mathcal{BA}$ and the time to broadcast m messages in the tree network.

Space Complexity. Algorithm $\mathcal{BA}$ uses two variables, c and m. Since m is used only to carry messages for the application level, the extra space used by our algorithm is only 1 bit.

Time Complexity. As seen in the proof of correctness of Algorithm $\mathcal{BA}$, any computation, starting from any initial configuration, is correct with respect to the specification $\mathcal{SP}_{\mathcal{BA}}$. Then, it is trivial to deduce the $O(1)$ stabilization time.

Broadcasting Time. We need to prove some properties to compute the time to broadcast messages. Next, We define what it means for a processor and a configuration to be *color synchronized*.

Definition 4.1 (Color Synchronization) *A processor $i \in V$ is color synchronized if at least one of the following conditions is true:* (i) $i \in (\{r\} \cup L)$, (ii) $c_i = c_{P_i}$*or* (iii) $\forall j \in Cld_i :: c_i = c_j$.

A configuration is color synchronized when all processors are color synchronized. We define $\mathcal{L}_{cs} \equiv$ Color Synchronized Configuration.

Lemma 4.4 *Let i be a processor such that i and its children are color synchronized. After a round, i is still color synchronized.*

Proof: Consider a processor $i \in I$. We do not need to consider the root and the leaf processors because they are always color synchronized by definition.

(i) Assume that i changes its $color$ in this round by copying its parent's color. By Property 3.1, the parent and children of i cannot execute any action during this round. Thus, i remains synchronized because Condition 2 of Definition 4.1 is satisfied.

(ii) Assume that i does not change its $color$ during a round. Let j be a child of i such that $c_j \neq c_i$ before the round (in a configuration, called γ). Since j is synchronized in γ, j must satisfy Condition 3 of Definition 4.1, i.e., all children of j have the same $color$ of j in γ. So, either $B2$ or $B3$ will be enabled at j. In this round, j will execute its action and c_j will become equal to c_i. Let k be a child of i such that $c_k = c_i$ before the round. k will not execute an action during this round. Condition 3 of Definition 4.1 at processor i is satisfied after the round.

(iii) Assume that P_i changes its $color$ during this round. By Property 3.1, i cannot execute any action during this round (see Case (i)).

□

Corollary 4.2 . $\forall \gamma \vdash \mathcal{L}_{cs} : \forall e \in \mathcal{E}_\gamma ::$ *the configuration reached after one round of computation starting from γ is also color synchronized.*

Lemma 4.5 *Starting from any arbitrary configuration, after $h-1$ rounds, the system will reach a color synchronized configuration.*

Proof: We prove the lemma by induction on h_i, the height of the subtree rooted at i,

(i) **Base Case:** $h_i = 0$. The lemma is true for $h_i = 0$ (the leaf processors) by definition.

(ii) **Hypothesis:** Assume that the lemma is true for $0 \leq h_i \leq m, m \leq h-2$. (Note that the lemma is true for $h_i = h$ because the root is always color synchronized.) Processors which are the root of a subtree of height less than or equal to m will be color synchronized within m rounds and will remain color synchronized thereafter. We now need to prove that the lemma is also true for $h_i = m+1$. Let i be a processor such that $h_i = m+1$. Assume that i is not color synchronized after m rounds and $\exists j \in Cld_i :: c_j \neq c_i$ before the $m+1^{\text{th}}$ round. i will not change its color during the $m+1^{\text{th}}$ round. If k is a child of i such that $c_k = c_i$ before the $m+1^{\text{th}}$ round, then k will not execute an action during the $m+1^{\text{th}}$ round. Let j be a child of i such that $c_j \neq c_i$ before the $m+1^{\text{th}}$ round. As, $h_j \leq m$, j will be color synchronized within m rounds by the hypothesis. All children of j will get the $color$ of j after m rounds. So, either $B2$ or $B3$ will be enabled at j. In the $m+1^{\text{th}}$ round, j will execute its action and c_j will become equal to c_i. Thus, after h_i rounds, all children of i share the color of i, and i is color synchronized.

□

Definition 4.2

$$\mathcal{L}^d_{even} \equiv \forall k \in]0,d] : \forall i \in V : \delta_i = 2k :: \ c_i = c_{P_i}$$

A configuration $\gamma \vdash \mathcal{L}^d_{even}$ if all processors at a distance $2,4,6,\ldots 2d$ from the root have the same $color$ as their parent.

$$\mathcal{L}^d_{odd} \equiv \forall k \in [0,d] : \forall i \in V : \delta_i = 2k+1 :: \ c_i = c_{P_i}$$

A configuration $\gamma \vdash \mathcal{L}^d_{odd}$ if all processors at a distance $1,3,5\ldots 2d+1$ from the root have the same $color$ as their parent.

$$\mathcal{G}^d_{even} \equiv \forall k \in]0,d] : \forall i \in V : \delta_i = 2k :: \ c_i \neq c_{P_i}$$

A configuration $\gamma \vdash \mathcal{G}^d_{even}$ if all processors at a distance $2,4,6,\ldots 2d$ from the root do not have the same $color$ as their parent.

$$\mathcal{G}^d_{odd} \equiv \forall k \in [0,d] : \forall i \in V : \delta_i = 2k+1 :: \ c_i \neq c_{P_i}$$

A configuration $\gamma \vdash \mathcal{G}^d_{odd}$ if all processors at a distance $1,3,5\ldots 2d+1$ from the root do not have the same $color$ as their parent.

$\mathcal{G}^0_{even} \equiv \mathcal{L}_{cs}$. $\mathcal{L}_{even} \equiv \mathcal{L}^d_{even}, 2d \geq h$. $\mathcal{L}_{odd} \equiv \mathcal{L}^d_{odd}, 2d+1 \geq h$. $\mathcal{G}_{even} \equiv \mathcal{G}^d_{even}, 2d \geq h$. $\mathcal{G}_{odd} \equiv \mathcal{G}^d_{odd}, 2d+1 \geq h$.

The following two properties follow from Algorithm $\mathcal{BA}$.

Property 4.1 *Let γ be a configuration such that $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^d_{odd} \wedge \mathcal{G}^d_{even}$. After one round of computation starting from γ, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^{d+1}_{even} \wedge \mathcal{G}^d_{odd}$.*

Property 4.2 *Let γ be a configuration such that $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^d_{even} \wedge \mathcal{G}^{d-1}_{odd}$. After one round of computation starting from γ, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^d_{odd} \wedge \mathcal{G}^d_{even}$.*

Lemma 4.6 *Starting from any arbitrary configuration $\gamma \in \mathcal{C}$, the system will reach a configuration $\gamma\prime$ in $h-1$ or h rounds such that $\gamma\prime \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$.*

Proof: By Lemma 4.5, all processors are synchronized (i.e., $\mathcal{L}_{cs}$ is true) within $h-1$ rounds. If all children of the root have the same $color$ as the root, then $\mathcal{L}^0_{odd}$ is true. Assume that there exists one child i of the root whose $color$ is not the same as the root. Since i is synchronized, all its children have its $color$ and B_2 is enabled at i. So, in the next round, i will execute B_2, copy the root's $color$, and $\mathcal{L}^0_{odd}$ will become true. □

Definition 4.3

$$\mathcal{L}_{oe} \equiv \mathcal{L}_{cs} \wedge ((\mathcal{L}_{odd} \wedge \mathcal{G}_{even}) \cup (\mathcal{L}_{even} \wedge \mathcal{G}_{odd}))$$

Lemma 4.7 *Starting from a configuration $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$, the system will reach a configuration $\gamma\prime$ in $h-1$ rounds such that $\gamma\prime \vdash \mathcal{L}_{oe}$.*

Proof: Let γ be a configuration satisfying $\mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd} \wedge \mathcal{G}^0_{even}$. Starting from γ, during the next round, the system will reach a configuration $\gamma_1 \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^1_{even} \wedge \mathcal{G}^0_{odd}$ (Property 4.1). Now, starting from γ_1, the system will reach a configuration $\gamma_2 \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^1_{odd} \wedge \mathcal{G}^1_{even}$ (Property 4.2). Thus, after $h-1$ rounds of computation starting from γ, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{oe}$. □

The following properties follow from Lemma 4.7 and Properties 4.1 and 4.2.

Property 4.3 *Starting from a configuration $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}_{even} \wedge \mathcal{G}_{odd}$, in one round, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{cs} \wedge \mathcal{L}_{odd} \wedge \mathcal{G}_{even}$. Starting from a configuration $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}_{odd} \wedge \mathcal{G}_{even}$, in one round, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{cs} \wedge \mathcal{L}_{even} \wedge \mathcal{G}_{odd}$.*

Property 4.4 *Starting from a configuration $\gamma \vdash \mathcal{L}_{oe}$, in alternate rounds, the processors at odd distance from the root (i.e., $\delta_i = 1, 3, \ldots$) and the processors at even distance from the root (i.e., $\delta_i = 0, 2, \ldots$), execute an action.*

Theorem 4.2 *Starting from a configuration $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$, a sequence of m messages broadcast by the root will reach all processors of the tree within $h + 2m - 1$ rounds.*

Proof: Assume that $\gamma \vdash \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$. The root can send its first message in γ. It takes h rounds after a message is broadcast by the root to reach all processors of the tree.

By Property 4.4, the root will be able to send a message once in every two rounds. Thus, starting from γ, the root will send the m^{th} message in $2m-1$th round and this last message will take another h rounds to reach all processors of the tree.

Thus, the maximum number of rounds necessary to broadcast m messages in the tree starting from γ is $h + 2m - 1$. □

Starting from an arbitrary configuration, it takes at most $2h+2m-1$ rounds for all processors to receive m messages broadcast by the root (Lemma 4.6 and Theorem 4.2).

5. Conclusion

We presented a new space efficient self-stabilizing synchronizing technique, the neighborhood synchronizer. This method implements the synchronization between a processor and its neighbors. Moreover, this scheme allows concurrency among processors which do not have a neighborhood relationship. The concurrency inherent in this scheme is similar to the pipelining scheme. We show an application of Algorithm $\mathcal{NS}$ to the design of an efficient broadcasting algorithm. Algorithm $\mathcal{BA}$ requires only $2h + 2m - 1$ rounds to broadcast m messages in the tree network.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[2] L. O. Alima, J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. In *ICDCS98 Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, 1998.

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

[4] B. Awerbuch. Complexity of network synchronization. *Journal of the Association of the Computing Machinery*, 32(4):804–823, 1985.

[5] 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.

[6] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[7] B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE*

Symposium on Foundations of Computer Science, pages 258–267, 1991.

[8] A. Bui, A. Datta, F. Petit, and V. Villain. Space optimal and fast self-stabilizing pif in tree networks. Technical Report RR 98-06, LaRIA, University of Picadie Jules Verne, 1998.

[9] J. Burns, M. Gouda, and R. Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[10] N. Chen, H. Yu, and S. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

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

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

[13] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[14] M. Gouda and F. Haddix. The linear alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 31–47. Carleton University Press, 1997.

[15] C. Johnen. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 125–140. Carleton University Press, 1997.

[16] H. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8:91–95, 1979.

[17] N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[18] M. Raynal and J. Helary. *Synchronization and Control of Distributed Systems and Programs*. John Wiley and Sons, Chichester, UK, 1990.

[19] G. Tel. *Introduction to distributed algorithms*. Cambridge university press, 1994.

[20] G. Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.