Parallel Processing Letters Vol. 12, Nos. 3 & 4 (2002) 327–340

OPTIMAL SNAP-STABILIZING NEIGHBOROOD SYNCHRONIZER IN TREE NETWORKS*

COLETTE JOHNEN
LRI - CNRS UMR 8623, Université Paris-Sud
France

LUC O. ALIMA
Unité d'Informatique, Université Catholique de Louvain
Belgium

AJOY K. DATTA
Department of Computer Science, University of Nevada Las Vegas
USA

and

SÉBASTIEN TIXEUIL
LRI - CNRS UMR 8623, Université Paris-Sud
France

Received September 1998
Revised May 2002
Accepted by S.K. Das

ABSTRACT

We propose a snap-stabilizing synchronization technique, called the *Neighborhood Synchronizer* ($\mathcal{NS}$) that synchronizes nodes with their neighbors in a tree network. The $\mathcal{NS}$ scheme has optimal memory requirement — only one bit per processor. $\mathcal{NS}$ is *snap-stabilizing* [11], meaning that it always behaves according to its specification. The proposed synchronizer being snap-stabilizing is optimal in terms of stabilization time. We show an application of the synchronizer by designing an efficient broadcast algorithm ($\mathcal{BA}$) in tree networks. $\mathcal{BA}$ is also snap-stabilizing and needs only $2h + 2m - 1$ rounds to broadcast m messages, where h is the height of the tree.

Keywords: Broadcasting, distributed algorithms, self-stabilization, synchronizer.

1. Introduction

Self-stabilization was introduced in distributed systems by Dijkstra in 1974 [14,15]. The paradigm of self-stabilization is considered to be 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

*A preliminary abstract of this paper was presented in [20].

guaranteed to converge to the intended behavior in finite time. The concept of *Snap-stabilization* was introduced in [10]. A *snap-stabilizing* algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps.

Designing synchronous protocols is simpler than designing asynchronous protocols. However it is more difficult to implement synchronous systems. A *synchronizer* [5] is a protocol which allows a synchronous protocol to run in an asynchronous system. Various types of synchronizers were developed in recent years. See [22], [25], and [26] for details.

Related Work. The research in the area of synchronizers started from the seminal work of Awerbuch [5]. However, the algorithms in [5] are not self-stabilizing. One approach to designing a self-stabilizing synchronizer is to combine the protocol of [5] with any self-stabilizing reset protocol [4,1,7]. Self-stabilizing synchronizers were proposed in [16,28,2] for tree networks, and [27,6,8] for general networks. PIF-based self-stabilizing synchronizers were proposed in [10,11,21] for tree networks, and [13,28] for general graphs. The algorithms in [10,11,13] are snap-stabilizing.

In [17], Gouda and Haddix proposed a self-stabilizing neighborhood synchronizer (which they referred to as alternator) for linear networks [17] and arbitrary networks [18]. Research on local mutual exclusion has been active in the recent years [9,24,23,3,19]. The solution to the local mutual exclusion problem can be used to design neighborhood synchronizers.

Our Contribution. We propose a snap-stabilizing synchronization technique, called the *Neighborhood Synchronizer* ($\mathcal{NS}$) that synchronizes nodes with their neighbors in a tree network. This scheme is optimal both in space and time. $\mathcal{NS}$ uses only one bit of memory per processor and is instantaneously stabilizing.

We then use the $\mathcal{NS}$ as a tool to design a very efficient snap-stabilizing broadcast algorithm in tree networks. The local synchronizer of [16] synchronizes only *two* neighboring processors, whereas $\mathcal{NS}$ synchronizes a processor with all its neighbors (parent and children in the tree network). The proposed broadcast algorithm needs only $2h + 2m - 1$ rounds to broadcast m messages. Any (non-self-stabilizing, self-stabilizing, or snap-stabilizing) PIF algorithm will take at least $\Omega(h \times m)$ rounds, to broadcast m messages,

Algorithm $\mathcal{NS}$ is also a solution to the local mutual exclusion problem [17].

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. Finally, we give the concluding remarks in Section 5.

2. Preliminaries

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, uses 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 assume that an underlying *local topology* maintenance protocol computes N_p. We also assume the existence of a *spanning tree* algorithm which maintains P_p. So, we consider N_p and P_p as constants in our algorithm. The height of a tree is denoted by h. h_p denotes the height of the subtree rooted at p. The distance of a processor p from the root r is denoted by δ_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. So, the variables of p can be accessed by p and its neighbors.

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

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

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.

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. 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 a step and a processor may take at most one step. This execution model is known as the *distributed daemon* [12]. The predicate $Enable\,(A, p, \gamma)$ is true if the guard of the action A is

true at processor p in the configuration γ. Similarly, the predicate $Enable(p, \gamma)$ is true 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 $\mathcal{E}$.

In order to compute the time complexity measure, we use the definition of *round* [16]. This definition captures the execution rate of the slowest processor in any computation. 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 the execution of one action (an action of the protocol or the disable action) 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.

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

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

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 (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:*

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

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

In this section, 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 Specification. We consider a computation e to satisfy the specification $\mathcal{SP}_{\mathcal{NS}}$ of Algorithm $\mathcal{NS}$ 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. Algorithm $\mathcal{NS}$

Informal Description. The main idea about the neighborhood synchronization is as follows: Every processor p uses a binary *color* variable, c_p to indicate the change of its state to its neighbors, an internal processor p changes c_p only when it finds that all its children have the same value as c_p and its parent has a different value than c_p. The protocol of the root and a leaf processor are similar except that the root (resp., a leaf processor) does not have to check the color of its (non-existent) parent (resp., children).

The Neighborhood synchronization can be used to simulate a reliable message passing mechanism using a register-based communication model as follows: The root sends a new message to its children and then waits until they read that new message. At that point, 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 the previous message. The leaves read a new message from their parent whenever the parent sends a new message. The root changes c_p to signal to its children that it has sent a new message. Similarly, the internal and leaf processors change their c to inform their parent (resp., children) that they have read (resp., hold) the previous (resp., new) message.

The Neighborhood Synchronizer algorithm $\mathcal{NS}$ is shown in Algorithm 1. Every processor i maintains a variable c_i, the state of i. We denote the set of children of i by Cld_i, *i.e.*, $Cld_i = N_i \setminus \{P_i\}$.

Algorithm 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}
$S_3 :: c_{P_i} \neq c_i \longrightarrow c_i := c_{P_i}$

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

We first prove the liveness of the algorithm. Then, we prove the correct behavior of the algorithm using the liveness result. The following properties follows directly from Algorithm $\mathcal{NS}$.

Property 1 $\forall \gamma \in \mathcal{C} : \forall i \in V :: Enable(i, \gamma) \Rightarrow (\forall j \in N_i :: \neg Enable(j, \gamma))$.

Informally, Property 1 states that if one particular processor is enabled in a configuration, then none of its neighbors is enabled in the same configuration (thus solving the local mutual exclusion problem).

Lemma 1 $\forall i \in (\{r\} \cup I) : \forall e \in \mathcal{E} : e = \gamma_1, \gamma_2, \ldots : \exists k \geq 1 : \forall j \in Cld_i :: c_j = c_i$ in γ_k.

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

Now consider the case where i never executes an action during a computation e (i.e., i never changes its color). If a child j of i has the color of i, then j cannot execute any action until i changes its color. We will prove this case by induction on the height of the subtree rooted at 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, $Enable(S_3, j, \gamma)$ will hold until j executes S_3. By fairness, j will eventually execute S_3 and $c_j = c_i$ becomes true. Following the same reasoning, all other children of i will eventually get the color of i.

- **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 gets the *color* of j. Then $Enable(S_2, j, \gamma)$ will be true and will remain true until j executes S_2 (by Property 1). By fairness, j will eventually execute S_2 and $c_j = c_i$ becomes true. If $c_j = c_i$, j cannot change its color. Similarly, all other children of i will get the color of i.

□.

Lemma 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 a configuration γ during a computation e. Let i be one of the processors nearest to the root among these processors. By Lemma 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.

1. Assume that $i = r$.

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

2. 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.

3. 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. (See Case 2.)

□.

Lemma 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:*

$$(S_i S_j)^\omega \cup (S_j S_i)^\omega$$

i.e., *between any two actions executed by a processor, all of its neighbors execute exactly one action.*

Proof. By Lemma 2, Processor i executes an infinite sequence of actions. Also, to be able to perform an action, i must be enabled, and when i executes an action, i becomes disabled. We consider two configurations γ_α and γ_β such that i executes an action between γ_α and $\gamma_{\alpha+1}$, and γ_β is the first configuration after $\gamma_{\alpha+1}$ where i is enabled again. Formally:

$$\begin{array}{l}\forall i \in V : \forall e \in \mathcal{E} : e = \gamma_1, \gamma_2, \ldots : \exists \alpha \geq 1, \exists \beta > \alpha + 1 :: \\ Enable(p, \gamma_\alpha) \wedge Enable(p, \gamma_\beta) \wedge (\forall \kappa \in]\alpha, \beta[: \neg Enable(p, \gamma_\kappa))\end{array}$$

What we need to prove is that between configurations $\gamma_{\alpha+1}$ and γ_β, every child of i executes exactly one action. Since i is disabled in $\gamma_{\alpha+1} \ldots \gamma_{\beta-1}$, any child of i can execute at most one action. i is enabled both in γ_α and γ_β, so every child of i executes at least one action between configurations $\gamma_{\alpha+1}$ and γ_β. By Property 1, the lemma follows. □.

Theorem 1 (Self-Stabilization) *Algorithm $\mathcal{NS}$ is a self-stabilizing neighborhood synchronizer algorithm.*

Proof. The theorem follows from Lemma 3 and the fact that we did not make any assumption on the initial configuration to prove Lemma 3. □.

Theorem 2 (Optimal Snap-Stabilization) *Algorithm $\mathcal{NS}$ is an optimal snap-stabilizing neighborhood synchronizer algorithm.*

Proof. Algorithm $\mathcal{NS}$ uses only one binary variable c. Thus, it requires only one bit. Since any computation starting from any initial configuration satisfies the specification, Algorithm $\mathcal{NS}$ has a zero stabilization time, meaning that it always satisfies its specification. □.

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

The root of a tree has an infinite sequence of messages to be broadcast to all processors of the tree. The root 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 has reached all processors of the tree. Thus, several messages may simultaneously be propagated down the tree, that is, effectively implementing a pipelining mechanism. We will show then the positive impact of the concurrent propagation of messages on the performance of Algorithm $\mathcal{BA}$.

Problem Specification. We consider a computation e to satisfy the specification $\mathcal{SP}_{\mathcal{BA}}$ of Algorithm $\mathcal{BA}$ if the following conditions are true:

1. Every message sent by the root is eventually received by all processors in the tree in the same order they were sent. We refer to this property as *Correct Delivery*.
2. All messages, except (possibly) the first δ_i messages, received by i were sent by the root. We call this property *Message Validity*.

We also require Algorithm $\mathcal{BA}$ to be self-stabilizing.

Informal Description. We make a few simple modifications in Algorithm $\mathcal{NS}$ to design Algorithm $\mathcal{BA}$ (shown in Algorithm 2). 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 from some application program and writes in m_r. The internal processors and leaf processors copy their parent's message from m_{P_i} into their own message variable, m_i.

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

Lemma 4 *$\forall i \in (\{r\} \cup I), \forall j \in Cld_i$, the messages sent by i are eventually received by j in the same order as they were sent with no loss or duplication.*

Proof. By Lemma 3 and Algorithm 2, after i receives a message, it cannot execute its action until all its children execute their action (*i.e.*, read the message from m_i). □.

Lemma 5 (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 and by using induction on the height of the tree. □.

Algorithm 2 ($\mathcal{BA}$) Broadcasting Algorithm for processor i.

Variable:
c_i: *color*
m_i: message
Constants:
Cld_i: set of children
P_i: parent
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}
$B_3 :: c_{P_i} \neq c_i \longrightarrow m_i := m_{P_i}$; $c_i := c_{P_i}$

Lemma 6 *$\forall i \in (I \cup L)$, all messages, except (possibly) the first one, received by i were sent by P_i.*

Proof. The first message received by i may not have been sent or received by P_i because the message may have been in transit due to some transient faults. □.

Lemma 7 (Message Validity) *All messages, except (possibly) the first δ_i messages received by i were sent by the root.*

Proof. The proof follows from Lemma 6 and by using induction on δ_i, the distance of i from the root. □.

Theorem 3 (Self-stabilization) *Algorithm $\mathcal{BA}$ is self-stabilizing.*

Proof. Follows from Lemmas 5 and 7, and the fact that these lemmas were proven independent of the initial configuration. □.

4.2. 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 one 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 0(1) stabilization time.

Theorem 4 (Optimal Snap-Stabilization) *Algorithm $\mathcal{BA}$ is an optimal snap-stabilizing broadcast algorithm.*

Proof. Algorithm $\mathcal{BA}$ uses only one additional binary variable c. Thus, it requires only one bit overhead. Since any computation starting from any initial configuration satisfies the specification, Algorithm $\mathcal{BA}$ has a zero stabilization time, meaning that it always satisfies its specification. □.

Broadcasting Time. We need to prove some properties to compute the time to broadcast messages.

Definition 3 (Color Synchronized Processor) *A processor $i \in V$ is color synchronized if at least one of the following conditions is true:*

1. $i \in (\{r\} \cup L)$.
2. $c_i = c_{P_i}$.
3. $\forall j \in Cld_i :: c_i = c_j$.

Definition 4 (Color Synchronized Configuration) *A configuration is color synchronized when all processors are color synchronized. We characterize any color synchronized configuration by a predicate, called $\mathcal{L}_{cs}$.*

Lemma 8 *Let i be a processor such that i and its children are colored 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.

1. Assume that i changes its *color* in this round by copying its parent's color. By Property 1, the parent and children of i cannot execute any action during this round. Thus, i remains synchronized because Condition 2 of Definition 3 is satisfied.

2. Assume that i does not change its *color* during a round.

   Let j be a child of i such that $c_j \neq c_i$ in a configuration γ before the round. Since j is synchronized in γ, j must satisfy Condition 3 of Definition 3, *i.e.*, all children of j must have the same *color* as j in γ. So, either $B2$ or $B3$ will be enabled at j. During the 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 the round. Condition 3 of Definition 3 at processor i is satisfied after the round.

3. Assume that P_i changes its *color* during this round. By Property 1, i cannot execute any action during this round (see Case 2).

□.

The following corollary follows directly from Lemma 8.

Corollary 1 $\forall \gamma \vdash \mathcal{L}_{cs}, \forall e \in \mathcal{E}_\gamma$, *any configuration reached after one round of computation starting from γ is also color synchronized.*

Lemma 9 *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,

1. **Base Case:** $h_i = 0$

   The lemma is true for $h_i = 0$ (the leaf processors) by definition.

2. **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 $m+1$ rounds, all children of i have the color of i, and i is color synchronized.

□.

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. Formally:

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

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

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

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

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

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

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

We also define the following notations:

$$\begin{array}{ll} \mathcal{G}^0_{even} \equiv \mathcal{L}_{cs} & \\ \mathcal{L}_{even} \equiv \mathcal{L}^d_{even}, 2d \geq h & \mathcal{L}_{odd} \equiv \mathcal{L}^d_{odd}, 2d+1 \geq h \\ \mathcal{G}_{even} \equiv \mathcal{G}^d_{even}, 2d \geq h & \mathcal{G}_{odd} \equiv \mathcal{G}^d_{odd}, 2d+1 \geq h \end{array}$$

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

Property 2 *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 3 *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 10 *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 9, 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. $\Box$.

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

Lemma 11 *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 2). 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 3). Thus, after $h-1$ rounds of computation starting from γ, the system will reach a configuration $\gamma\prime \vdash \mathcal{L}_{oe}$. $\Box$.

The following properties follow from Lemma 11 and Properties 2 and 3.

Property 4 *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 5 *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.*

Lemma 12 *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 for the message to reach all processors of the tree. By Property 5, the root will be able to send a message once in every two rounds. Thus, starting

from γ, the root will send the mth 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 $h+2m-1$. □.

Theorem 5 *Starting from an arbitrary configuration, it takes at most* $2h+2m-1$ *rounds for all processors to receive* m *messages broadcast by the root.*

Proof. Follows from Lemmas 10 and 12. □.

5. Conclusions

We presented a new space efficient self-stabilizing synchronizing technique, the neighborhood synchronizer. This method implements the synchronization between a processor and its neighbors. This scheme also 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 $\mathcal{NS}$ by extending it into an efficient broadcasting algorithm. Algorithm $\mathcal{BA}$ requires only $2h+2m-1$ rounds to broadcast m messages in the tree network.

Both algorithms are optimal in space. $\mathcal{NS}$ makes use of only one bit of memory, and $\mathcal{BA}$ add only one bit overhead to the message size. Both algorithms are also optimal in time because they are snap-stabilizing. The delay (see [13]) of $\mathcal{NS}$ and $\mathcal{BA}$ are zero and $O(h)$ rounds, respectively.

6. 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 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] S. Antonoiu and P. K. Srimani. Self-stabilizing protocol for mutual exclusion among neighboring nodes in a tree structured distributed system. *Parallel Algorithms and Applications*, 14(1):1–18, 1999.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer LNCS:1914*, pages 223–237, 2000.

[10] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-stabilizing pif algorithms in tree networks without sense of direction. In *Proceedings of SIROCCO'99, Carleton University Press*, pages 32–46, 1999.

[11] A. Bui, A. K Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 78–85. IEEE Computer Society, 1999.

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

[13] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Snap-stabilizing pif algorithm in arbitrary networks. In *Proceedings of ICDCS'02*. IEEE Computer Society Press, 2002.

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

[15] S. Dolev. *Self-stabilization*. The MIT Press, 2000.

[16] 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.

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

[18] M. G. Gouda and F. Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53. IEEE Computer Society, 1999.

[19] S. T. Huang. The fuzzy philosophers. In *Proceedings of Workshop on Advances of Parallel and Distributed Computational Models*, pages 130–136. LNCS 1800, 2000.

[20] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems*, pages 487–494, 1999.

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

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

[23] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[24] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer LNCS:1693*, pages 254–268, 1999.

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

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

[27] G. Varghese. Self-stabilization by local checking and correction (Ph.D. thesis). Technical Report MIT/LCS/TR-583, MIT, 1993.

[28] 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.