Self-stabilization with causally synchronized wave on tree network

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

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

Abstract

In this paper, we focus on the description of a technique called "causally synchronized waves", defined on a tree networks. This technique is a self-stabilizing one, meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state), the network is guaranteed to converge to a proper execution.

As an application of our method, an efficient self-stabilizing broadcast algorithm is provided: only $O(1)$ memory space (in addition to the tree structure maintenance) is required at each processor, this algorithm will stabilize in $O(h)$ "rounds" (where h is the height of the tree network), only $h+1+2n$ rounds are needed to broadcast n messages in the whole network.

Keywords: distributed algorithm, self-stabilization, waves chain, propagation with feedback, broadcast algorithm.

Résumé

Dans ce papier, nous présentons une technique appelée "vagues causalement synchronisées" qui est définie dans les réseaux arborescents. Cette technique est auto-stabilisante, ce qui signifie qu'à partir de n'importe quel état (obtenu après une perturbation qui modifie la mémoire) le système va automatiquement converger vers un état cohérent.

Un algorithme de diffusion est proposé comme application de notre technique: seulement $O(1)$ bits est nécessaire à l'algorithme. Le temps de stabilisation est $O(h)$ pas de calcul (h étant la hauteur de l'arbre) et la diffusion dans le réseau de n messages demande seulement $h+1+2n$ pas de calcul.

Mots clés: algorithme distribué, auto-stabilisation, train de vagues, propagation avec feedback, algorithme de diffusion.

1 Introduction

As distributed systems are getting larger and larger, thus leading to more possible malfunctions in some of their base element, the issue of *fault-tolerance* is becoming of primary importance. Two approaches are possible to achieve the fault-tolerance property in distributed system. One, called "robust", uses redundancy to mask the effect of faults. The other one, known as *self-stabilization* [Dij74], allows systems to temporarily present an "abnormal" behavior. After any unexpected perturbation modifying the memory state, and leaving the whole system in an arbitrary configuration, the system is allowed to exhibit "faulty" behavior for a finite time only, then it must converge to a proper execution. Such a property is very desirable for any distributed system, because self-stabilizing systems cope with memory corruptions, and with processors (or processes)[1], and communication-channels crashing and recovering (i.e. dynamic networks). Furthermore, self-stabilizing systems do not require particular initial state when a processor recovers assuming that processor codes are not corrupted. In order to build self-stabilizing systems, a number of paradigm and techniques have been designed, such as [GM91, Sch93, BGM93, DH95].

Chang [Cha82] and Segall [Seg83] have defined the concept of propagation with feedback (also called wave propagation): the initiator of propagation wave will receive a notification of the propagation termination. Broadcast with feedback have been used intensively in distributed computing ([Awe88], [AG94], [APSV91]). For instance, the synchronizers (β and γ) presented by Awerbuch in [Awe85], use the technique of propagation with feedback to transform any synchronous protocol into a version for asynchronous network in the message passing model. A self-stabilizing version of the synchronizer γ was presented by Awerbuch and Varghese in [AV91]. In a complete propagation, all processors of the network are involved in the ebb and flow wave. We define the concept of *causally synchronized wave*, which is a partial and causally synchronized propagation with feedback. In a causally synchronized d-wave, only processors at distance up to d of the wave initiator are involved. Some processors that are involved in waves initiated by other processors initialize their own waves. These processors synchronize both wave chains. Therefore wave chains are causally synchronized. A causally synchronized d-waves chain provides a simple global synchronizer ([ABDT97]), when d is equal to height of the tree network. It also provides a broadcast algorithm where at the end of a wave, the tree root is ensured that all processors have get the broadcasted data. Unfortunately, this broadcast schema is very inefficient because the initiator needs to wait all processor feedbacks before beginning a new broadcast. We present, in this paper, a very efficient broadcast algorithm as an application of the causally synchronized 1-wave technique. This algorithm requires only $O(1)$ memory space (in addition to the tree structure maintenance) at each processor; It will stabilize in $O(h)$ rounds (where h is the height of the tree network); and only $h + 1 + 2n$ rounds are needed to broadcast n messages in the whole network. Although our algorithm relies upon an underline tree network topology it is not less general than the algorithm in [AEYH92] since a spanning tree of a network can be obtained by a number of self-stabilizing algorithms ([AKY90], [CYH91], [HC92], [SS92], [DIM93],[TH94], [Dol93], [BLB95], and [AB97]) whose a memory efficient algorithm requiring only $O(1)$ bits of memory per incident network edge in [Joh97]. Notice that our algorithm requires less memory space that the self-stabilizing algorithm on general network presented in [AEYH92] (where the size of the messages is $0(log(N.\Delta))$ bits - N being the network size and Δ being processor's degree -).

The remainder of the paper is organized as follows. In Section 2 we present the distributed systems we consider and define self-stabilization in that context. In Section 3, we define the causally synchronized d-wave. parent. A self-stabilization solution to the causally synchronized

[1] We assume that each processor runs exactly one process.

1-wave problem is presented in Section 3.2. We present a self-stabilizing broadcast algorithm as application of this solution in section 4. The correctness of the broadcast algorithm is proven in section 4.2; and in section 4.3, the performance of the algorithm is established.

2 Model

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

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

A link connecting processor i to processor j is uniquely identified by the two-tuple (i, j), and for every $(i, j) \in E$, processors i and j are called *neighbors*.

We use only tree networks in this paper. The *root* processor of a system $\mathcal{S}$ is denoted by $\mathcal{S}^r$, the set of *leaf* processors by $\mathcal{S}^l$, and the set of other processors (called hereafter the *intermediate* processors) by $\mathcal{S}^s$. So, the set of all processors is $\{\mathcal{S}^r\} \cup \mathcal{S}^s \cup \mathcal{S}^l = V$. The parent of processor i is denoted by P^i, and the set of children of processor i by Cld^i. The height of a system $\mathcal{S}$ is denoted by $h(\mathcal{S})$.

The height of a particular processor i (*i.e.*, its distance from the root $\mathcal{S}^r$) is denoted as $\overline{i}$, and the set of processors that are up to distance d from processor i is denoted by $\overline{V}_i^d$. For the sake of convenience, we denote the set of processors that are up to distance d from the root as $\overline{V}^d$. The set of processors that are exactly at distance d of the root is denoted by $V(d)$.

Communications. 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 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.

States and Configurations. The *state* of processor is defined by the values of its local variables and field variables. A *configuration* of a distributed system $\mathcal{S} = (V, E)$ is a product of the states of all processors of $\mathcal{S}$. The set of configurations of $\mathcal{S}$ is denoted as $\mathcal{C}$.

Actions and Computations. Each processor executes an algorithm. The algorithm consists of a set of variables and a finite set of actions. 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 algorithm of i is a boolean expression involving the local variables of i, the field variables of i, and its neighbors. The statement of an action of i updates some local variables and/or field variables of i. 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.

During a *computation step*, one or more processors execute a step, and a processor may take at most one atomic step; this is known as the *distributed daemon* [BGM89]. A *round* is a maximal computation step where all processors that hold the guard of an action, execute the corresponding action during the step.

A *computation* e of an algorithm $\mathcal{A}$ is a *fair, maximal* sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration c_{n+1} is reached from c_n by a computation step. c_1 is called the *initial configuration* of e. *Fairness* of the sequence means that if a processor may continuously perform an action along a sequence of $\mathcal{A}$, it will eventually perform an action. *Maximum* means that the sequence is either infinite, or it is finite, and no action of $\mathcal{A}$ 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 an algorithm $\mathcal{A}$ in system $\mathcal{S}$ starting with a particular initial configuration $c \in \mathcal{C}$ is denoted by $\mathcal{E}_c$. The set of computations of $\mathcal{A}$ in system $\mathcal{S}$ whose initial configurations are elements of $B \subset \mathcal{C}$ is denoted as $\mathcal{E}_B$. $\mathcal{E}$ is the set of all possible fair and maximal computations ($\mathcal{E} = \mathcal{E}_{\mathcal{C}}$).

Predicates. Let $\mathcal{X}$ be a set. $x \triangleleft P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$. We distinguish a special predicate: true (satisfied by each element of $\mathcal{X}$) formally defined as follows : $x \triangleleft \mathsf{true}$.

Definition 2.1 (Attractor) *Let B_α and B_β be two predicates defined on $\mathcal{C}$ of system $\mathcal{S}$ that performs an algorithm $\mathcal{A}$. B_α is an attractor for B_β if and only if the following condition is true:*

$$\forall b_\beta \triangleleft B_\beta : \ \forall e \in \mathcal{E}_{b_\beta} : \ (e = c_1, c_2, \ldots) :: \ \exists n \geq 1, c_n \triangleleft B_\alpha$$

Definition 2.2 (Self-stabilization) *The algorithm $\mathcal{A}$ is self-stabilizing for the specification predicate $\mathcal{SP}$ on $\mathcal{E}$, if the following condition holds: $\forall e \in \mathcal{E}$: $(e = c_1, c_2, \ldots) :: \ \exists n \geq 1, \forall e_n \in \mathcal{E}_{c_n} : e_n \triangleleft \mathcal{SP}$*

Informally, an algorithm $\mathcal{A}$ is self-stabilizing for $\mathcal{SP}$ if only if any execution of $\mathcal{A}$ has a suffix that is a correct execution. The temporal activities of a self-stabilizing algorithm can be divided into three phases:

- The *fault* phase: The period during which faults may occur in the system. These faults may corrupt the volatile memory of processors and links.
- The *stabilizing* phase: The period during which the system may not exhibit the correct behavior. However, no external faults may occur.
- 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 algorithm can be measured in time and memory needed to achieve the stabilization.

Definition 2.3 (Space complexity) *The space complexity of a self-stabilizing algorithm is the number of local states of a processor performing the self-stabilizing algorithm.*

Definition 2.4 (Time complexity) *The time complexity of a self-stabilizing algorithm is the maximal number of rounds needed to reach a configuration from which all executions satisfy the specification predicate, after the faults cease to occur.*

3 Causally synchronized wave for tree networks

In this section, we define the scheme of causally synchronized waves. After stating formally the definition of causally synchronized d-wave (section 3.1), we describe and prove our self-stabilizing solution in the case where $d = 1$, in section 3.2.

3.1 Causally synchronized wave chain.

propagation and feedback Informally, a d-wave of communication ($d \geq 1$) is a wave propagation from a processor $\mathcal{I}^r$ to every other processors at distance at most d from $\mathcal{I}^r$, followed by a feedback from every reached processor to $\mathcal{I}^r$. The feedback is an acknowledgment sent back from these processors to $\mathcal{I}^r$. So at the end of the wave, the initiator knows that processors at distance d of it have been reached by the wave. The waves are causally synchronized : processors at distance d from $\mathcal{I}^r$ will start d-waves synchronized with the waves initiated by $\mathcal{I}^r$. These processors initiate their waves when they are reached by the wave initiated by $\mathcal{I}^r$. Thus the propagation is done in complete sub-tree rooted to $\mathcal{I}^r$ through a cascade of waves, although the initial wave does not reach all processors in $\mathcal{I}^r$ sub-tree.

As the system is asynchronous, some processors may perform faster that others, thus the feedback phase may start in some parts of the network while other parts of the network are still performing the propagation phase.

In order to define formally a causally synchronized d-wave, we consider four kinds of processors: $\mathcal{I}^r$, which may not be the root, is the initiator of the d-wave. $\mathcal{I}^s$ is the set of processors at distance (strictly) less that d from $\mathcal{I}^r$ and that have children (the intermediates). $\mathcal{I}^l$ is the set of processors in $\overline{V}^d_{\mathcal{I}^r} \setminus (\{\mathcal{I}^r\} \cup \mathcal{I}^s)$ that does not have any child (the leaves). $\mathcal{I}^f$ contains the processors of $\overline{V}^d_{\mathcal{I}^r} \setminus (\{\mathcal{I}^r\} \cup \mathcal{I}^s)$ that have children. The processors of $\mathcal{I}^f$ are simultaneously initiators of waves and involved in the waves initiated by $\mathcal{I}^r$ (*i.e.* at distance d from an $\mathcal{I}^r$).

Color. It is needed in addition to distinguish communication waves from each other within a wave chain. That way, one may assume that all causally synchronized d-waves are colored, by adding color to processors participating in the wave. The color of processor i will be noted $i|_c$. This color may take values in $\{black, white\}$.

The only processors that need to check the color of their parent and the color of their children to evaluate one action guard are processors of $\mathcal{I}^f$ because they synchronize waves.

Processor i being in propagation phase is noted $i\downarrow$, and processor i being in feedback phase is noted $i\uparrow$,

Definition 3.1 (Causally Synchronized Wave) *A causally synchronized d-wave initiated by $\mathcal{I}^r$, is a finite minimal sequence of configurations $c_1, \ldots, c_k$ such that:*

1. *In c_1, $\mathcal{I}^r\downarrow \wedge\ \mathcal{I}^r|_c = \neg b \wedge\ \forall i \in \mathcal{I}^s \cup \mathcal{I}^l$, $i\uparrow \wedge\ i|_c = \mathcal{I}^r|_c$.*
2. *$\forall n \in [1, k[$, c_{n+1} is reached from c_n with one or more of the following actions:*
   (a) *$\mathcal{I}^r|_c$ moves to $\neg\mathcal{I}^r|_c$ if and only if $\forall j \in Cld^{\mathcal{I}^r}$, $j|_c = \mathcal{I}^r|_c \wedge\ j\uparrow$.*
   (b) *$j \in \mathcal{I}^s$ moves from $j\uparrow$ to $j\downarrow$ and $j|_c$ moves from $\neg b$ to b if and only if $P^j\downarrow$ and $P^j|_c \neq j|_c$.*
   (c) *$j \in \mathcal{I}^s$ moves from $j\downarrow$ to $j\uparrow$ if and only if $\forall j' \in Cld^j$, $j|_c = j'|_c \wedge\ j'\uparrow$.*
   (d) *$j \in \mathcal{I}^l$, $j|_c$ moves from $\neg b$ to b if and only if $P^j\downarrow$ and $P^j|_c \neq j|_c$.*

(e) $j \in \mathcal{I}^f$, $j|_c$ *moves from* $\neg b$ *to* b *if and only if* $P^j \downarrow$ *and* $P_c^j = \neg h$ *and* $\forall j' \in Cld^j$, $j|_c = j'|_c \wedge\ j' \uparrow$.

3. *In* c_k, $\mathcal{I}^r \downarrow \wedge\ \mathcal{I}^r|_c = b \wedge\ \forall i \in \mathcal{I}^s \cup \mathcal{I}^l,\ i \uparrow \wedge\ i|_c = \mathcal{I}^r|_c$.

A causally synchronized d-wave initiated by $\mathcal{I}^r$ *is denoted by* $W(d, \mathcal{I}^r)$.
The color of the wave $W(d, \mathcal{I}^r)$ *is the color of* $\mathcal{I}^r$ *in the last configuration of the wave. It is denoted by* $W(d, \mathcal{I}^r)|_c$.

Observation 3.1 *During a causally synchronized wave, every processor of* $\overline{V}^d_{\mathcal{I}^r}$ *changes its color exactly once.*

With these definitions, we state the specification of causally synchronized wave chain in a distributed system working on a tree network.

Definition 3.2 (Causally Synchronized Wave Chain) *k causally synchronized d-waves initiated by* $\mathcal{I}^r$ $(W(d, \mathcal{I}^r)_1, W(d, \mathcal{I}^r)_2, \ldots W(d, \mathcal{I}^r)_k)$ *are color synchronized if and only if the following conditions are verified:*

1. $\forall n \in [1, k[,\ W(d, \mathcal{I}^r)_n|_c \neq W(d, \mathcal{I}^r)_{n+1}|_c$,
2. $\forall n \in [1, k],\ W(d, \mathcal{I}^r)_n|_c \in \{black,\ white\}$.

k causally synchronized d-waves initiated by $\mathcal{I}^r$ *are denoted by* $WC(k, d, \mathcal{I}^r)$.

Observation 3.2 $\forall n \in [1, k[$, *the last configuration of* $W(d, \mathcal{I}^r)_n$ *is the first configuration of* $W(d, \mathcal{I}^r)_{n+1}$.

When $d \leq h(\mathcal{S}) - 1$, some processors are wave initiators and receivers. These processors synchronize the two related wave chains.

Theorem 3.1 *Let* w *be a leaf of the subtree of height* d_y *rooted at* y.

$$WC(n-1, d_w, w) \subset WC(n, d_y, y)$$

Proof : w is an element of $\mathcal{I}^f$. The action (e) performed by w during a wave initiated by y is also the action (a) of a wave initiated by w. Each wave begins with an action (a). During a causally synchronized wave rooted at y, w performs an action (e); and thus starts a causally synchronized wave rooted at w such that

$$W(d_w, w)_1|_c = W(d_y, y)_1|_c$$

During $WC(n, d_y, y)$, w has started n causally synchronized wave chains, thus it has finished at least $n-1$ causally synchronized wave chains. □

3.2 Local Synchronization

We present an algorithm which is an implementation of a solution to the problem specified in Definition 3.2 where $d = 1$. The algorithm provides a "local synchronizer" because once stabilized, the actions of a processor will alternate with one and only one action of each processor's neighbor.

Formal description. Our technique uses one binary variable per processor: c, the color of the processor. It may take the values *black* or *white* and may be negated (assuming $\neg$ *white* = *black*, and that $\neg$ *black* = *white*).

Moreover, a processor has access to the set of its children (denoted by `Cld`) and to its parent (denoted by `P`). For each variable v of processor i, referencing this variable is denoted by i_v. One should note that no *propagation/feedback* variable is needed. The initiator of a wave is always in the propagation phase for its children, and the children of an initiator are always in feedback phase in regard of the initiator.

According to Algorithm 3.1, the root of the system executes action $\mathcal{A}_1$ –implementation of action(a)–, the leaves execute action $\mathcal{A}_3$ –implementation of the action (d)–, while the intermediate processors execute action $\mathcal{A}_2$ – implementation of action (e). Each processor will always perform the same action. In this case, $\mathcal{I}^s$ is empty therefore there is not implementation of the actions (b), and (c).

A system where each processor performs the algorithm 3.1, is such that each processor performs an infinity of causally synchronized 1-waves. In the following, such a system is denoted as $\mathcal{LS}$.

Algorithm 3.1 Local synchronizer algorithm

Variables:

c is the color of the processor

Constants:

`Cld` denotes the set of children of the processor,

`P` denotes the parent of the processor.

{*This action is executed by the root processor only*}
$\mathcal{A}_1$: $(\forall j \in \texttt{Cld},\ j_c = c) \longrightarrow c := \neg c$.

{*This action is executed by the intermediate processors only*}
$\mathcal{A}_2$: $\texttt{P}_c \neq c \wedge\ (\forall j \in \texttt{Cld},\ j_c = c) \longrightarrow c := \texttt{P}_c$.

{*This action is executed by the leaf processors only*}
$\mathcal{A}_3$: $\texttt{P}_c \neq c \longrightarrow c := \texttt{P}_c$.

3.2.1 Local synchronizer correctness

We first show that within finite time, any non-root processor will color synchronize with its parent.

Lemma 3.1 $\forall i \in V \setminus \{\mathcal{LS}^r\}$, *if* i *is enabled, then* $\forall j \in \{P^i\} \cup Cld^i$, j *is not enabled.*

Proof : The proof follows directly from observation of Algorithm 3.1. □

Lemma 3.2 $\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, $(e = c_1, c_2, \ldots)$, $\forall i \in \mathcal{LS}^s \cup \mathcal{LS}^l$, $\exists n \geq 1$, $i_c = P_c^i$ *in* c_n.

Proof : We prove the lemma by induction on the height of the tree.
Base case. Let us assume that there exists some processor $i \in \mathcal{LS}^l$ such that i has never the color of P^i during a computation, called e. All along e, the action $\mathcal{A}_3$ is and stays enabled until i performs it because P^i cannot change its color (perform an action) due to lemma 3.1. By fairness, whatever the current state, and the computation performed, i will eventually execute the action $\mathcal{A}_3$. After the action : $i_c = P_c^i$.

Induction step. Let us assume that all processors at a distance inferior to n to a leaf will have the color of their parent whatever the computation performed. In turn, let us assume that there exists some processor i at distance $n+1$ from a leaf such that i has never the color of P^i during a computation, called e. Along e, P^i may not change its color since at least one of its children does not have its color; i never performs the action $\mathcal{A}_2$ (changes its color) otherwise the computation would reach a state where $i_c = P^i_c$. Along e, once a child of i, called j, has the color of i; j cannot perform any action.

All children of i are at distance n of a leaf, thus eventually all of them will get the color of their parent, during e. Therefore, the system will reach a configuration where i may perform the action $\mathcal{A}_2$: all children of i have the color of i, and $i_c \neq P^i_c$. Then along e, i will always be able to perform this action. By fairness, i will eventually execute it. After the action : $i_c = P^i_c$. □

Now, we demonstrate the liveness of $\mathcal{LS}$ system.

Lemma 3.3 (liveness) $\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, $\forall j \in \{\mathcal{LS}^r\} \cup \mathcal{LS}^s \cup \mathcal{LS}^l$, *$j$ performs an action infinitely often.*

Proof : Let i be a processor that does not perform an action infinitely often during a computation, called e. Let i' be the nearest processor of $\mathcal{LS}^r$ that does not perform an action infinitely often during e. During e, at some point i' never performs an action (never changes its color). Then, all children of i' will get the color of i' (lemma 3.2). As i' cannot change its color, all children of i' will keep the color of i'.

- If $i' = \mathcal{LS}^r$, i' holds the guard of the action $\mathcal{A}_1$ forever. By fairness, i' will eventually execute the action $\mathcal{A}_1$.
- If $i' \neq \mathcal{LS}^r$ and $i'_c = P_c$. Processor $P^{i'}$ will eventually performs an action, and changes its color because according to the hypotheses, i' is the nearest processor of $\mathcal{LS}^r$ that does not perform an action infinitely often.

In any case, i' will reach a state in e where $i'_c \neq P_c$. Processor $P^{i'}$ may not change its color since at least one of its children does not have its color. Thus i will always be able to perform the action $\mathcal{A}_2$. By fairness, i will eventually perform it. □

Lemma 3.4 (synchronization) *Let $i \in V$ and let $j \in Cld^i$. Let $\mathcal{A}^i$ denote an action executed at processor i. Let e be a fair and maximal computation. The projection of e on the actions of i and j is:*

$$\left(\mathcal{A}^j \cup \epsilon\right)\left(\mathcal{A}^i\mathcal{A}^j\right)^\omega$$

Proof : After $\mathcal{A}^i$, j does not have the same color as i. i may not perform an action before the color changing of j. After $\mathcal{A}^i$, j executes an action once. Then its action is disabled while i has not changed its color (*i.e.* executed $\mathcal{A}^i$).

Since i and j change their color infinitely often (from Lemma 3.3), any projection on i and j of a computation is of the form:

$$\left(\mathcal{A}^j \cup \epsilon\right)\left(\mathcal{A}^i\mathcal{A}^j\right)^\omega$$

□

A direct consequence of the lemma 3.4 is that between two actions of a processor i, all its neighbors perform exactly one action. Thus, $\mathcal{LS}$ synchronizes locally processors.

3.2.2 Local synchronizer performance

Definition 3.3 (Synchronization) *A processor i is unsynchronized if and only if the following properties are verified:*

1. *$i \in V \setminus \{\mathcal{LS}^r\}$ (i is not the tree root)*
2. *$i_c \neq P_c^i$ (i has not the color of its parent)*
3. *$\exists j \in Cld^i$, $i_c \neq j_c$ (at least one children of i has not its color).*

Lemma 3.5 . *Starting from any initial configuration c, $\forall j \in \overline{V}^n$, $(1 \leq n \leq h(\mathcal{LS}))$, if j is synchronized in c, j remains synchronized.*

Proof : Let i be a synchronized processor. Three kinds of actions are possible:

1. i changes its color by taking its parent's color (it remains synchronized)
2. its parent changes its color. Thus during this round, i did not perform any action (lemma 3.1). Some children of i may have the color of i before the round; these children did not perform any action during the round (they still have the color of i after the first round). The other ones hold the guard of $\mathcal{A}_2$ or $\mathcal{A}_3$, before the round : as they are synchronized and they do not have the color of their parent, all their children have their colors. According to the definition of round, they perform an action during the round and change their color.

   Thus, at the end of the round, all children of i have the color of i : i remains synchronized.
3. a child of i changes its color. But in this case, the child takes the color of i (and i stays synchronized).

In any case, i remains synchronized.

□

Lemma 3.6 *Starting from any configuration, after $h(\mathcal{LS}) - 1$ rounds, all processors are synchronized.*

Proof : The proof is by induction on the distance of processors to the leaves.
Base case. Let i be a processor at distance $h(\mathcal{LS}) - 1$ from the root. The children of i are the leaves. After the first round, i has the color of its parent : i is synchronized. Or i does not have the color of its parent. Thus, i did not perform any action during the first round. Some children of i have the color of i before the first round; these children did not perform an action during the first round (still have the color of i after the first round), the other ones hold the guard of the $\mathcal{A}_3$, before the first round. -Thus they have performed this action during the first round (according to the definition of a round). Thus after the first round, all children of i have its color. In all cases, processors at distance $h(\mathcal{LS}) - 1$ from the root are synchronized, after the first round.
Induction step. Let us assume that after $n < h(\mathcal{LS})$ rounds, all processors at distance superior of $h(\mathcal{LS}) - n$ from the root are synchronized. They will stay synchronized (lemma 3.5). Let i be a processors at distance $h(\mathcal{LS}) - n - 1$ of the root; that is not synchronized after the n rounds : some children of i does not have the color of i. Since, these processors are synchronized (they are at distance $h(\mathcal{LS}) - n$ from the root), all their own children have their color : they hold $\mathcal{A}_2$ guard, and within the $n + 1^{\text{th}}$ rounds, they take the color of their parent. □

Lemma 3.7 *Let $\mathcal{REG}_1$ be the set of configurations where all processors are synchronized. $\mathcal{REG}_1$ is an attractor.*

Proof : The proof follows directly from the lemma 3.5 and 3.6 □

Definition 3.4

$$\mathcal{L}^l_{even} \equiv \left\{ c \in \mathcal{C} | \forall i \in V(2k)\ k \in]0,l], i_c = P^i_c \right\}$$

In a configuration of $\mathcal{L}^l_{even}$, all processors at a distance 2, 4, 6 ... , or 2l from the root have the color of their parent.

$$\mathcal{L}^l_{odd} \equiv \left\{ c \in \mathcal{C} | \forall i \in V(2k+1)\ k \in [0,l], i_c = P^i_c \right\}$$

In a configuration of $\mathcal{L}^l_{odd}$, all processors at a distance 1, 3, 5 ... , or 2l+1 from the root have the color of their parent.

$$\mathcal{G}^l_{even} \equiv \left\{ c \in \mathcal{C} | \forall i \in V(2k)\ k \in]0,l], i_c \neq P^i_c \right\}$$

In a configuration of $\mathcal{G}^l_{even}$, **no** *processor at a distance 2, 4, 6 ... , or 2l from the root have the color of their parent.*

$$\mathcal{G}^l_{odd} \equiv \left\{ c \in \mathcal{C} | \forall i \in V(2k+1)\ k \in [0,l], i_c \neq P^i_c \right\}$$

In a configuration of $\mathcal{G}^l_{odd}$, **no** *processor at a distance 1, 3, 5, ... , or 2l+1 from the root have the color of their parent.*

Notation 3.1

If $2l \geq h(\mathcal{LS})$, *then* $\mathcal{L}_{even} = \mathcal{L}^l_{even} = \mathcal{L}^h_{even}$ $\mathcal{G}_{even} = \mathcal{G}^l_{even} = \mathcal{G}^h_{even}$

If $2l+1 \geq h(\mathcal{LS})$, *then* $\mathcal{L}_{odd} = \mathcal{L}^l_{odd} = \mathcal{L}^h_{odd}$ $\mathcal{G}_{odd} = \mathcal{G}^l_{odd} = \mathcal{G}^h_{odd}$

Observation 3.3 $\mathcal{G}^0_{even} = \mathcal{C}$

Lemma 3.8 *Starting from any configuration, after $h(\mathcal{LS}) - 1$ or $h(\mathcal{LS})$ rounds, $\mathcal{L}^0_{odd}$ is satisfied and all processors are synchronized.*

Proof : After $h(\mathcal{LS}) - 1$ rounds, all processors are synchronized (lemma 3.7). Either all children of the root have its color (then $\mathcal{L}^0_{odd}$ is satisfied), or some children of the root do not have its color. Since these processors are synchronized, all their own children have their color. Then they hold $\mathcal{A}_2$ guard; and within the next round, they take the root color. Thus after $h(\mathcal{LS}) - 1$ or $h(\mathcal{LS})$ rounds, $\mathcal{L}^0_{odd}$ is satisfied. □

Lemma 3.9 *Let c be a configuration such that $c \in \mathcal{L}^l_{odd} \cap \mathcal{G}^l_{even}$ and all processors are synchronized. The processors of $V(2k)$ $(\forall k \in [0,l])$ will perform an action during the round performed from c. Let c' be the configuration reached from c after a round. We have $c' \in \mathcal{L}^{l+1}_{even} \cap \mathcal{G}^l_{odd}$.*

Proof : The following two properties are verified $\forall k \in [0,l]::\ \forall i \in V(2k)$:

1. $\forall j \in Cld^i$, $i_c = j_c$ $(c \in \mathcal{L}^l_{odd})$

2. $i_c \neq P_c^i$ $(c \in \mathcal{G}_{even}^l)$.

The root holds the guard of $\mathcal{A}_1$; and the processors in $V(2k)$ $(\forall k \in]0, l])$ hold the guard of an action in c. During the next round, they will perform an action (according to the definition of a round). At the end of the round, the following two properties are verified $\forall k \in [0, l]:: \forall i \in V(2k)$:

1. $\forall j \in Cld^i$, $i_c \neq j_c$ $(c' \in \mathcal{G}_{odd}^l)$
2. $i_c = P_c^i$ $(c' \in \mathcal{L}_{even}^l)$.

In c', the processors at distance $2l+1$ do not have the color of their parent. As they are synchronized (lemma 3.5), their children (if they exists) have their color. Thus $\mathcal{L}_{even}^{l+1}$ is satisfied in c'. □

Lemma 3.10 *Let $c \in \mathcal{L}_{even}^l \cap \mathcal{G}_{odd}^{l-1}$ be a configuration such that all processors are synchronized in c. The processors in $V(2k+1)$ $(\forall k \in [0, l-1])$ will perform an action during the round performed from c. Let c' be the configuration reached from c after one round. Then $c' \in \mathcal{L}_{odd}^l \cap \mathcal{G}_{even}^l$.*

Proof : The following two properties are verified $\forall k \in [1, l]:: \forall i \in V(2k-1)$:

1. $\forall j \in Cld^i$, $i_c = j_c$ $(c \in \mathcal{L}_{even}^l)$
2. $i_c \neq P_c^i$ $(c \in \mathcal{G}_{odd}^{l-1})$.

The processors in $V(2k-1)$ hold the guard of an action in c. During the next round, they will perform an action. At the end of the round, the following two properties are verified $\forall k \in [1, l]::$ $\forall i \in V(2k-1)$:

1. $\forall j \in Cld^i$, $i_c \neq j_c$ $(c' \in \mathcal{G}_{even}^l)$
2. $i_c = P_c^i$ $(c' \in \mathcal{L}_{odd}^{l-1})$.

In c' the processors at distance $2l$ do not have the color of their parent. To stay synchronized, their children (if they exists) have their color (lemma 3.5). Thus $\mathcal{L}_{odd}^l$ is satisfied in c'. □

Theorem 3.2 *$\mathcal{REG}_2 = \mathcal{REG}_1 \cap ((\mathcal{L}_{odd} \cap \mathcal{G}_{even}) \cup (\mathcal{L}_{even} \cap \mathcal{G}_{odd}))$. $\mathcal{REG}_2$ is an attractor.*

Proof : Starting from any configuration, after $h(\mathcal{LS}) - 1$ or $h(\mathcal{LS})$ rounds, $\mathcal{L}_{odd}^0$ is satisfied; and all processors are synchronized (lemma 3.7).

Let $c1$ be a configuration such that $c1 \in \mathcal{L}_{odd}^0 \cap \mathcal{G}_{even}^0$; and all processors are synchronized. During the next round, the processor of $V(0)$ $(\mathcal{LS}^r)$ will perform the action$\mathcal{A}_1$; the configuration reached from $c1$, $c2$ is such that : $c2 \in \mathcal{L}_{even}^1 \cap \mathcal{G}_{odd}^0$ (lemma 3.9). All processors are synchronized in $c2$ (lemma 3.5). The processors of $V(1)$ will perform the action $\mathcal{A}_2$ during the round performed from $c2$. Let $c3$ be the configuration reached from $c2$ after a round. We have $c3 \in \mathcal{L}_{odd}^1 \cap \mathcal{G}_{even}^1$ (lemma 3.10). All processors are synchronized in $c3$.

Thus, we conclude that from $c1$, in $h(\mathcal{LS})$ rounds, the configuration reached (c) belongs to $\mathcal{REG}_2$. We also conclude that from c, alternatively, during one round, the processors of $V(2k)$ $(\forall k \in [0, h(\mathcal{LS})/2])$ will perform an action; then during the next round, the processors of $V(2k+1)$ will perform an action (lemma 3.9 and 3.10). □

3.2.3 Complexity

In this section, we study complexity measures for the $\mathcal{LS}$ system. Definitions for this measures are given in definition 2.3 and 2.4.

Proposition 3.1 (Space complexity) *The space complexity for the $\mathcal{LS}$ system is $O(1)$ at each processor ($O(n)$ for the whole network).*

Proof : Algorithm 3.1 only need one field variable : the color. The color variable may take values in the set of symbolic constants $\{black, white\}$. The number of state of a processor is 2. The space complexity at each processor is $O(1)$. □

Proposition 3.2 (Time complexity) *The time complexity for the $\mathcal{LS}$ system is $O(h)$.*

Proof : The proof follows directly from the proof of the theorem 3.2. □

4 Broadcast : Application of $\mathcal{LS}$

In this section, we use Algorithm 3.1 as a basis for building a self-stabilizing broadcast algorithm.

4.1 Broadcast algorithm

Informal problem specification. The root processor of the tree is fed with a number of *messages* which are to be sent to all other processors in the network. From this informal definition, it follows that *all* processors must receive *all* messages.

A classic way of performing such a broadcast is to use propagation of information with feedback: the root processor emits a message, then wait for all processors (not only its children) an acknowledgment of the received message, before sending the next message (if any).

We use a different and more efficient approach here since the root processor only waits for its children acknowledgment for sending a new message. Then, in stabilized phase, several messages may propagate down the tree at the same time.

Informal solution. Every processor in the network is given an m variable, large enough to contain any single message the root may wish to send. At the root processor is added a helper function GetNewMessage that takes no arguments and returns the next message to be sent.

Then the actions of Algorithm 3.1 need to be modified as follows:

1. The *root* processor changes its message m using GetNewMessage when all its children have the same color as its own one. If GetNewMessage returns $\emptyset$, the root does not change its color.

2. The *intermediate* processors copy their parent's message P_m and change their color when their parent's color has changed (*i.e.* is different from their own color) and all their children color is the same as theirs.

3. The *leaf* processors copy their parent message P_m when taking their parent's color.

Formal description. According to Algorithm 4.1, the root processor executes action $\mathcal{A}'_1$, the leaf processors execute action $\mathcal{A}'_3$, while the intermediate processors execute action $\mathcal{A}'_2$.

Algorithm 4.1 Broadcast algorithm using the $\mathcal{LS}$ mechanism

Variables:

c is the color of the processor,
m is the message to be broadcasted,

Constants:

`Cld` denotes the set of children of the processor,
`P` denotes the parent of the processor.

{*This action is executed by the root processor only*}
$\mathcal{A}'_1$: $(\forall j \in \texttt{Cld},\ j_c = c) \longrightarrow m := \mathsf{GetNewMessage};\quad c := \neg c.$

{*This action is executed by the intermediate processors only*}
$\mathcal{A}'_2$: $\texttt{P}_c \neq c \wedge\ (\forall j \in \texttt{Cld},\ j_c = c) \longrightarrow m := \texttt{P}_m;\quad c := \texttt{P}_c.$

{*This action is executed by the leaf processors only*}
$\mathcal{A}'_3$: $\texttt{P}_c \neq c \longrightarrow m := \texttt{P}_m;\quad c := \texttt{P}_c.$

4.2 Broadcast correctness

One may informally define the legal behavior of a broadcast algorithm: all broadcasted messages issued by the root will be received (in the same order, with no lost nor duplication) by all processors in the tree. However, in the context of self-stabilizing systems, legitimate configurations can not be properly defined because a single configuration may seem correct while the messages stored in the processor were not actually sent by the root. No information in a single configuration allows an observer from the outside to decide if a message stored by a processor is erroneous.

Therefore, our broadcast specification is given in terms of computation predicates instead of configuration predicate.

In the following, we consider the $\mathcal{BS}$ system, where each processor performs Algorithm 4.1.

Definition 4.1 (Message Reception) *A processor i receives a message if and only if it executes its enabled action.*

Lemma 4.1 *Let $i \in V$, $\forall j \in Cld^i$, any message received by i is eventually received by j in the same order, with no loss nor duplication.*

Proof : From Lemma 3.4, after i has received one message, all of its children receive exactly one message (the same message). There is no loss nor duplication nor de-sequencing since i may not receive another message before all its children have received the first message. □

Theorem 4.1 *Let $i \in V$, any message received by i is eventually received by all processors in the subtree rooted at i in the same order, with no loss nor duplication.*

Proof : The proof is by induction on h_i the height of the subtree rooted at i.
Base case. The base case ($h_i = 1$) is proved by Lemma 4.1.
Induction step. Let's assume that all processors at distance n from i receive all messages received by i in the same order, without loss nor duplication (*i.e.* the property is true with $h_i = n$). In turn, from Lemma 4.1, their children receive these messages, in the same order, with no loss nor duplication (*i.e.* the property is true with $h_i = n + 1$). □

Lemma 4.2 *Let $i \in V \setminus \{\mathcal{BS}^r\}$, after one reception at i, all subsequent messages received by i have been received by P^i.*

Proof : The proof is a direct consequence of Lemma 3.4. □

Note that the first received message by i may have not been received by P^i.

Theorem 4.2 *Let $i \in V(d_i)$, after n receptions ($1 \leq n \leq d_i$), all messages received by i have been received by the ancestor of i at distance n.*

Proof : The proof is by induction on n.
Base case. The base case is proved by Lemma 4.2.
Induction step. The ancestor of i at distance n is noted i_n. After n receptions, all messages received by i have been received by i_n (induction hypotheses). Messages are received by i in the same order, without loss nor duplication (from Lemma 4.1) that i_n has sent them. Thus the $n+1$ first message received by i was also received by i_n. If this message is the first message received by i_n then it may not have been received by i_{n+1} (the parent of i_n). The following messages will been received by i_n and i_{n+1} : because they are not the first message received by i_n. □

4.3 Broadcast performance

Theorem 4.3 *Starting from any initial configuration c, and considering any possible $e \in \mathcal{E}_c$, $\mathcal{BR}^r$ will need at most $h(\mathcal{BR})$ rounds in e to correctly broadcast its current message. Then, the broadcast of $n+1 > 0$ additional messages from $\mathcal{BR}^r$ takes $h(\mathcal{BR}) + (2n) + 1$ rounds in e.*

Proof : Whatever the configuration initial and the computation performed, after $h(\mathcal{BR})$ or $h(\mathcal{BR}) - 1$ rounds, $\mathcal{L}^0_{odd}$ is satisfied (from Lemma 3.8).

Thus the root performs its action (begin the broadcast of a message) within the first $h(\mathcal{BR})^{\text{th}}$ rounds, or during the $(h(\mathcal{BR})+1)^{\text{th}}$ round. From this point, the root performs its action (broadcast a message) each two rounds (proof of the theorem 3.2). Therefore, the root needs $(2n)+1$ rounds, to begin the broadcast of n successive messages.

A message broadcasted needs $h(\mathcal{BR})$ rounds to reach the leaves. During the first round, the children of the root receive the message; during the second round, the processors at distance 2 of the root receive the message; After $h(\mathcal{BR})$ rounds, all processors have received the message. Thus after $h(\mathcal{BR}) + (2n) + 1$ rounds, all messages have being broadcasted by the root; and the last message has been received by each processor.

From theorem 4.1, all processors receive the n broadcasted messages in the same order, without loss or duplication. Thus, after $h(\mathcal{BR}) + (2n) + 1$ rounds, the n messages issued from the root have being received by each processor. □

We can now state about a bound on the stabilization time of our algorithm: the time needed so that any message received by a processor has effectively been previously sent by the root.

Corollary 4.1 *Starting from any initial configuration c, and considering any possible $e \in \mathcal{E}_c$, after $2h(\mathcal{BR}) + 1$ rounds the system is stabilized.*

Proof : Within $h(\mathcal{BR})+2$ rounds, the root has broadcasted its first message. Thus after $2h(\mathcal{BR})+1$ rounds, the message broadcasted by the root has been received by all processors. In addition, processors receive the broadcasted messages in the same order, with no loss nor duplication □

5 Conclusion

In the design of distributed algorithms for various applications several very general problems for distributed networks appear frequently as subtasks. These elementary tasks include broadcasting (e.g. propagation of a reseting or a termination message), achieving global synchronization between processors, triggering the execution of some event in each processor, ... Causally synchronized wave is a generic construction which once parameterized yields a solution to these problems. A self-stabilizing global synchronizer in [ABDT97], and a efficient self-stabilizing broadcast algorithm (presented in this paper) is obtained as direct application of causally synchronized wave schema.

The obtained broadcast algorithm propagates messages in a pipeline way : a processor only ensure the propagation of the message to its children. The cooperation of all processors allows the broadcasted messages to reach every processor. That way, the obtained algorithm is optimal in space and time.

References

[AB97] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA97)*, pages ??–??, 1997.

[ABDT97] L. O. Alima, J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. unpublished, 1997.

[AEYH92] E Anagnostou, R El-Yaniv, and V Hadzilacos. Memory adaptive self-stabilizing protocols. In *WDAG92 Distributed Algorithms 6th International Workshop Proceedings, Springer-Verlag LNCS:647*, pages 203–220, 1992.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[AKY90] 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.

[APSV91] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 32th Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[AV91] B Awerbuch and G Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 32th Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *JACM*, 32(4):804–823, 1985.

[Awe88] Baruch Awerbuch. On the effects of feedback in dynamic network protocols. *FOCS88 Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 231–245, 1988.

[BGM89] JE Burns, MG Gouda, and RE Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[BGM93] JE Burns, MG Gouda, and RE Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7:35–42, 1993.

[BLB95] F Butelle, C Lavault, and M Bui. A uniform self-stabilizing minimum diameter tree algorithm. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972*, pages 257–272, 1995.

[Cha82] E. Chang. Echo algorithm : Depth parall operations on general graphs. *IEEE Transactions on Software Engineering*, 9:504–512, 1982.

[CYH91] NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[DH95] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, 1995.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[Dol93] S Dolev. Optimal time self-stabilization in dynamic systems. In *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer-Verlag LNCS:725*, pages 160–173, 1993.

[GM91] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[HC92] ST Huang and NS Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[Joh97] 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.

[Sch93] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[Seg83] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29:23–35, 1983.

[SS92] S Sur and PK Srimani. A self-stabilizing distributed algorithm for BFS spanning trees of a symmetric graph. *Parallel Processing Letters*, 2:171–179, 1992.

[TH94] MS Tsai and ST Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4:65–72, 1994.