Self-stabilizing Space Optimal Synchronization Algorithms on Trees

Doina Bein, Ajoy K. Datta, and Lawrence L. Larmore

University of Nevada, Las Vegas, USA
{siona, datta, larmore}@cs.unlv.edu

Abstract. We present a space and (asymptotically) time optimal self-stabilizing algorithm for simultaneously activating non-adjacent processes in a rooted tree (Algorithm $\mathcal{SSDST}$). We then give two applications of the proposed algorithm: a time and space optimal solution to the local mutual exclusion problem (Algorithm $\mathcal{LMET}$) and a space and (asymptotically) time optimal distributed algorithm to place the values in min-heap order (Algorithm $\mathcal{HEAP}$). All algorithms are self-stabilizing and uniform, and they work under any unfair distributed daemon. In proving the time complexity of the heap construction, we use the notion of *pseudo-time*. Pseudo-time is similar to *logical time* introduced by Lamport [12].

Keywords: heap, local mutual exclusion, self-stabilization.

1 Introduction

Fault-tolerance is the ability of a system to withstand transient faults. A fault-tolerant system is guaranteed to continue to perform its function when a number of transient errors has occurred. In 1973 [8], Dijkstra defined a distributed system to be *self-stabilizing* when, "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps."

Self-stabilizing algorithms aim to achieve performance comparable to that of non-stabilizing distributed algorithms when transient faults or arbitrary initialization cause the system to enter a state where a non-stabilizing algorithm cannot continue to perform its task properly. In this paper, we propose a general synchronization scheme for a rooted tree, and use this scheme to solve two fundamental problems: heap construction and local mutual exclusion.

Related Work. The self-stabilizing heap problem has been studied in [1, 4, 5, 10, 13]. The first self-stabilizing binary-search tree construction algorithm was proposed in [4]. In [1], the self-stabilizing algorithm for a min-heap construction improves the algorithm of [5] in three ways: no global reset is required, the time complexity is reduced from $O(nh)$ to $O(h)$ (h is the height of the tree with n nodes), and the space complexity per node is reduced from $O(degL)$ to $O(deg + L)$ (deg is the degree of the process and L is the maximal size of the initial values in the tree). Synchronization among the nodes is achieved by using the global rooted synchronizer defined in [2], plus two additional bits. In

[13], the self-stabilizing max-heap protocol that uses a neighborhood synchronizer protocol [11] reduces the memory requirement further to $2L + 3$ bits; its time complexity is $O(h)$. A heap construction that supports insert and delete operations in arbitrary states of a variant of the standard binary heap [7] with capacity K is proposed in [10]. It takes $O(m \log K)$ heap operations to stabilize (m is the initial number of items in the heap). The space complexity per node i is $O(h_i)$, where h_i is the height of the subtree T_i rooted at i.

Bein *et al.* [4] proposed the first snap-stabilizing binary search tree (BST) and the first snap-stabilizing heap construction algorithm. (A snap-stabilizing algorithm is a self-stabilizing algorithm with stabilization time of 0 rounds). The algorithms use a PIF scheme [6] to synchronize the nodes in the tree. The space complexity of the snap-stabilizing heap construction algorithm is $3L + 3$.

Contributions. We propose a space and (asymptotically) time optimal self-stabilizing algorithm for simultaneously activating non-adjacent processes in a rooted tree (Algorithm $\mathcal{SSDST}$). It uses $1+\lceil log(deg)\rceil$ bits in each node (deg is the node degree); during the first $2h + 2t - 1$ rounds, every node is enabled at least t times, *i.e.*, on the average, once every second round. For a synchronous system, after at most $2h$ steps, every node is enabled every second step. If the synchronous network starts in a normal starting configuration, then a node is active every other step from the beginning.

We then give two applications on rooted trees of the proposed algorithm: a time and space optimal solution to the local mutual exclusion problem (Algorithm $\mathcal{LMET}$), and a space and (asymptotic) time optimal solution to the heap problem (Algorithm $\mathcal{HEAP}$). Algorithm $\mathcal{LMET}$ uses only $2+\lceil log(deg)\rceil$ bits per node and stabilizes in 0 rounds (it is *snap*-stabilizing). During the first $2h+2t-1$ rounds, a node enters its CS at least t times. Algorithm $\mathcal{HEAP}$ arranges n values, not necessarily distinct, in non-decreasing order from top to bottom (min-heap), in at most $4(7h/2 - 4)$ rounds (h = height). Each process holds only one value at any moment, and uses a total of $1+\lceil log(deg)\rceil$ bits per node, not counting the bits needed to store the value being sorted (deg = node degree) which is optimal, thus an improvement over [13, 4].

In proving the time complexity of heap-building, we use the notion of *pseudo-time.* Each node in the network has a "local clock" which has the property that when any action must be executed between the node and its children, the local clocks of all the nodes involved in the action have the same value.

Outline of the Paper. In Section 2, we briefly introduce self-stabilization and the topological models used by the proposed algorithms. Section 3 contains a description of Algorithm $\mathcal{SSDST}$, followed by a sketch of its proof of correctness. Algorithm $\mathcal{LMET}$ is presented in Section 4. In Section 5 we first present a min-heap algorithm for an abstract model of communication (Algorithm $\mathcal{A\_HEAP}$), and then show how the min-heap will be built using the usual shared-memory model of communication (Algorithm $\mathcal{HEAP}$). A sketch of correctness proof of Algorithm $\mathcal{A\_HEAP}$ is given in 5.3. The reduction of Algorithm $\mathcal{A\_HEAP}$ to Algorithm $\mathcal{HEAP}$ is given in 5.4. We finish with concluding remarks in Section 6.

2 Computational Models

We consider an asynchronous, rooted tree of n processors, with height h. The root node is denoted by R. We assume that an underlying self-stabilizing spanning tree construction protocol maintains the parent pointer p_v and the set of children D_v of a node v. For the root node R, $p_R = \perp$. For a leaf node v, $D_v = \perp$.

If the topology of the network that is given as the input to the spanning tree construction algorithm changes, the spanning tree may change. This will change the input to our protocols (local mutual exclusion and heap). In that sense, the proposed protocols can deal with dynamic trees. The model of communication among the neighboring nodes is shared memory — a process can read and write its own memory, but can only read the memory of its neighbors.

The program of every processor consists of a finite set of guarded actions of the form: $<$ *label* $>::<$ *guard* $>\rightarrow<$ *action* $>$, where each guard is a function of the variables of the processor and its direct neighbors. The *state* of a process is defined by the values of its variables. The *system state* (*configuration*) is the Cartesian product of all the nodes' states. If an action has its guard, a Boolean expression, evaluated to *true*, then it is called *enabled.* A node with at least one enabled guard is called *enabled.* A daemon will non-deterministically select a non-empty subset of enabled nodes to execute one of its enabled actions. Guard evaluation and execution of the its action are done in one atomic step.

We assume an asynchronous system. In order to compute the time complexity, we use the definition of *round* [9]. A round is a minimal sequence of computation steps during which each processor that was enabled in the first configuration of the sequence executes at least once during this sequence.

We consider the strongest distributed daemon, the *unfair* daemon. The *unfair daemon* does not have a fairness mechanism: a continuously enabled process will not necessarily be selected for execution unless it is the only enabled process.

Let $\mathcal{C}$, the set of all possible states, and a predicate $\mathcal{P}$ over $\mathcal{C}$. We denote by $\mathcal{L}_\mathcal{P} \subseteq \mathcal{C}$ the set of all *legitimate states with respect to* $\mathcal{P}$. Let $\mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{C}$. $\mathcal{C}_2$ is a *closed attractor* for $\mathcal{C}_1$ if (i) every execution starting in $\mathcal{C}_1$ eventually reaches a configuration in $\mathcal{C}_2$, and (ii) every execution starting in $\mathcal{C}_2$ remains in $\mathcal{C}_2$.

Definition 1 (Self-stabilization). *If $\mathcal{P}$ is a predicate, a protocol S is called* self-stabilizing to $\mathcal{P}$ *if $\mathcal{L}_\mathcal{P}$ is a closed attractor for $\mathcal{C}$.*

3 Self-stabilizing Distributed Simultaneous Execution of Non-adjacent Nodes in a Rooted Tree $\mathcal{SSDST}$

Each node v holds a variable $S \in \{A, B\}$ and a pointer $i \in 0..|D_v| - 1$ to some child of v. Thus, the total memory requirement of node v is $1 + \lceil \log{(deg)} \rceil$ bits (deg is the node degree). (For a binary tree, Algorithm $\mathcal{HEAP}$ uses at most three bits per node.)

For simplicity we write $S = S.v$. The predicate *check*(v, s) means that the node v exists and has the value s for its variable S. Let *execute*(v) denote a generic action.

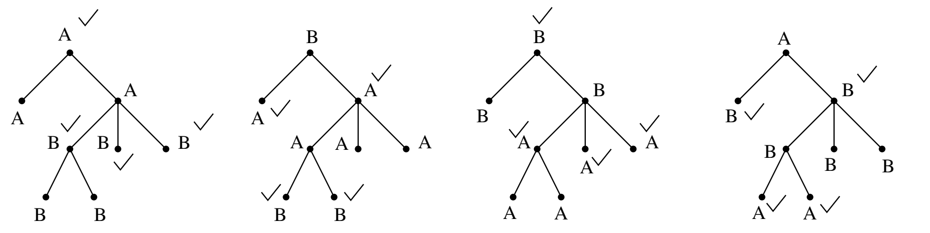
Algorithm 3.1. *Algorithm $\mathcal{SSDST}$*

```
Predicate $check(v, s) \equiv (v = \bot \vee S.v = s)$

Actions for any node $v$
ABB  $S{=}B \wedge check(p_v, A) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], B) \longrightarrow execute(v)\ ;\ S{=}A$
BAA  $S{=}A \wedge check(p_v, B) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], A) \longrightarrow execute(v)\ ;\ S{=}B$
```

Actions ABB and BAA are enabled at node v when the following two conditions are true: (i) either it has no parent, or its parent's S-value is different from its S-value, and (ii) all its children's S-values are the same as its S-value.

For example, given a network of eight nodes starting in a so-called normal starting configuration (Figure 1(a)), the only enabled nodes are of even depth (the root and the children of the root's children). If we assume a synchronous system, the next execution step brings the system into the configuration in Figure 1(b), in which the only enabled nodes are of odd depth. The next configuration is shown in Figure 1(c), followed by the one in Figure 1(d). Then the system returns to the configuration illustrated in Figure 1(a). The cycle repeats forever.



(a) A configuration (b) After one step (c) After two steps (d) After three steps

Fig. 1. Four steps in a synchronous system

3.1 Proof of Correctness for $\mathcal{SSDST}$

In this section, we show that Algorithm $\mathcal{SSDST}$ stabilizes in at most $2h+2k-1$ rounds, to the global predicate

k-Exec: $\equiv \{\forall$ node v, v has executed macro *execute* at least k times $\}$

and works under the unfair distributed daemon.

We extend the notions of *configuration-string* and *difference-string* to the tree network. We show that in every configuration, during execution of $\mathcal{SSDST}$:

- No node is enabled if any of its neighbors is enabled (local mutual exclusion) (Property 1)
- At least one node is enabled (no deadlock); after it executes, a node becomes disabled until all its neighbors execute (Property 2)
- During the first $2h + 2k - 1$ rounds every node executes at least k times (no starvation) (Lemma 1).

We then show that $\mathcal{SSDST}$ works under the unfair distributed daemon (Property 3, Section 3.2).

Henceforth, $n > 1$, as the case $n = 1$ is trivial. Let the *configuration tree* be the tree in which every node is represented by its S-value only.

A *normal starting configuration* is a configuration in which each branch of the configuration tree is a prefix of $(AABB)^n$ (the string of length $4n$ obtained by concatenating $AABB$ n times). Starting from a normal starting configuration, the enabled nodes are alternately of even and odd depth (Figure 1). The *binary edge labeling* is the labeling where an edge between nodes with the same S value is labeled 0 and other edges are labeled 1.

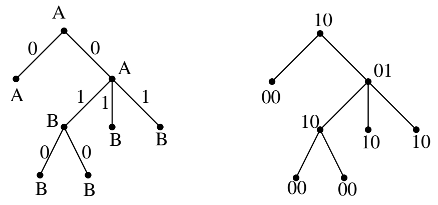
Definition 2. *Given a configuration tree C, we let DT_C, the* difference tree*, be the tree in which every node v is represented by a two-bit string $DT_C(v) = b_0b_1$ such that:*

$$b_0 = \begin{cases} 1, \text{ if } p_v = \bot \text{ or the link } (p_v, v) \text{ is labeled } 1 \\ 0 \text{ otherwise} \end{cases}$$

$$b_1 = \begin{cases} 1, \text{ if } \exists w \in D_v \text{ s.t. the link } (v, w) \text{ is labeled } 1 \\ 0 \text{ otherwise} \end{cases}$$

If C is understood, write DT instead of DT_C. Given a binary edge labeling and the S-value of some node, the corresponding configuration tree C is uniquely defined. Given a difference tree DT and the S-value of some node, the corresponding configuration tree C is uniquely defined.

For example, for the configuration in Figure 1(a), the binary edge labeling is given in Figure 2(a) and the difference tree is given in Figure 2(b).



(a) Binary edge labeling (b) Difference tree

Fig. 2. Some configuration

Given any configuration tree C, a node v is enabled if and only if $DT_C(v) = 10$.

Property 1. For any configuration tree C and for any node v, if node v is enabled to execute, then no neighbor of v is enabled.

Property 2. (i) In any configuration tree C there exists at least one enabled node.

(ii) For any node v, if node v is enabled and is selected to execute, then after the execution is completed, its actions are disabled.

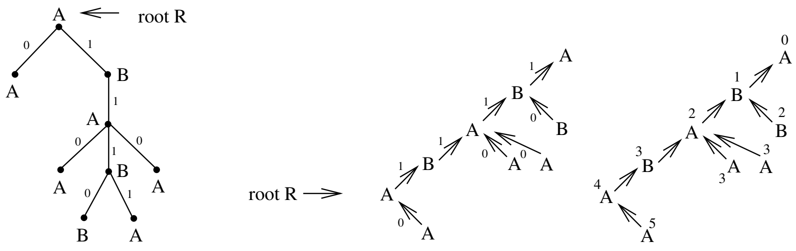
Given a node v and its parent p_v where $S.p_v = a$ and $S.v = b$, the notation "$a \leftarrow b$" denotes that state b does not block state a from being enabled (for p_v

to be enabled in state a, $S.v$ must be b). The notation $a \rightarrow b$ indicates that state a does not block state b from being enabled (for v to be enabled in state b, $S.p_v$ needs to be a).

We use the above notation to define *layers* as follows. We start defining the layers of nodes from node R and going down the tree until we reach the leaf nodes. Node R is placed on some layer. If node v is an internal node on a certain layer, then for any child node $w \in D_v$:

- if $S.v \rightarrow S.w$ then w is one layer higher
- if $S.v \leftarrow S.w$ then w is one layer lower.

We can represent a configuration tree using this notation in a level ordering, where the peak nodes are the enabled nodes. The binary edge labeling is consistent with the orientation of the arrows between a node and its parent, and a node and its children (1 for $\nearrow$, 0 for $\nwarrow$). For example, the sawtooth-like arrangement of the configuration tree in Figure 3(a) is given in Figure 3(b).



(a) A configuration tree (b) Layered arrangement (c) Delay values

Fig. 3. Calculating the delay values

Definition 3 (Node Delay). *For each node v we define delay$[v]$ to be a non-negative integer characterized as follows: (i) there exists at least one node whose delay is* 0*, and (ii) if delay$[u] = d$ and node v is a neighbor of node u such that $S.v \rightarrow S.u$ then delay$[v]$ = delay$[u] + 1$, and (iii) if $S.v \leftarrow S.u$ then delay$[v]$ = delay$[u] - 1$.*

The delay of some node is in fact the layer on which the node is arranged in the layered arrangement.

The delay values of the nodes in Figure 3(a) are given in Figure 3(c). An enabled node has all the adjacent arrows pointing towards it. For a tree of height h, for any node v, *delay*$[v]$ is a value between 0 and $2h$. The number of rounds that a node waits before it becomes enabled cannot exceed its delay value.

Let d_0 be the array of the delay values in the starting configuration and D_0 be the maximal value of d_0 over all nodes: $1 \leq D_0 \leq 2h$.

Lemma 1. *For any node v and any value $t > 0$ node v executes t times within the first $d_0[v] + 2t - 1$ rounds.*

Proof. We define the predicate $\mathcal{P}(q)$ as follows: For any node v, for any $t \geq 1$, node v executes t times within the first q rounds if $q \geq d_0[v] + 2t - 1$.
For any $q \geq 1$, Predicate $\mathcal{P}(q)$ holds (induction on q).

Corollary 1. *For any node v and any value $t > 0$ node v executes t times within the first $2h + 2t - 1$ rounds.*

Proof. Follows from Lemma 1: for any node v, $2h \geq d_0[v]$.

3.2 The Unfair Distributed Daemon

In this section we show that Algorithm $\mathcal{SSDST}$ works under the unfair distributed daemon. A sufficient condition to prove that a certain algorithm works under the unfair daemon is to show that a continuously enabled node which is never selected eventually becomes the only enabled node. If a node v is enabled to execute but not selected by the distributed daemon, it remains enabled. Since the unfair daemon must select a non-empty subset of the enabled nodes in every computation step, it will be forced to select v (Property 3).

Property 3. If a node v is enabled to execute but is not selected by the daemon, it remains enabled until it gets selected. Every continuously enabled node will be eventually selected by the unfair distributed daemon.

4 Self-stabilizing Local Mutual Exclusion Algorithm on Rooted Trees $\mathcal{LMET}$

Each node holds three variables: variable S that takes values in the set $\{A, B\}$, a pointer $i \in 0..|D_v| - 1$ to some child of v, and a Boolean variable *request* that is *true* whenever the process requests access to its critical section CS. Thus, the total memory requirement of node v is $2 + \lceil \log(deg) \rceil$ bits per node (deg is the node degree).

For some node v, let $S = S.v$ and $request = request.v$. The predicate $check(v, s)$ is defined in Section 3.

A protocol solves the local mutual exclusion problem if any configuration of the system running the protocol has two properties ([3]): *(i) safety* - no two neighboring nodes can be simultaneously enabled to execute their critical sections (CS), and *(ii) liveness* - a node requesting to execute its CS will eventually do so.

Algorithm 4.1. *Algorithm $\mathcal{LMET}$*

```
Actions for any node $v$
ABB  $S = B \wedge check(p_v, A) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], B) \longrightarrow$
            if $request$ then $CS$; $request = false$
            $S = A$
BAA  $S = A \wedge check(p_v, B) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], A) \longrightarrow$
            if $request$ then $CS$; $request = false$
            $S = B$
```

Property 1 shows that $\mathcal{LMET}$ has the safety property. Lemma 1 shows that $\mathcal{LMET}$ has the liveness property.

5 Self-stabilizing Min-heap Algorithms for a Rooted Tree

In this section we present two algorithms for min-heap problem in a rooted tree: $\mathcal{A\_HEAP}$ (Section 5.1), and $\mathcal{HEAP}$ (Section 5.2). Algorithm $\mathcal{A\_HEAP}$ is implemented in an abstract model. Algorithm $\mathcal{HEAP}$ is implemented in the shared-memory model.

Let x and y be two values to be swapped. Swapping can be done in three steps without using an extra variable, as follows:

1. $x = x + y$ 2. $y = x - y$ 3. $x = x - y$

Alternatively, we could use "$\oplus$, bit-wise exclusive or, instead of addition and subtraction.

5.1 Heap Construction in a Rooted Tree

Algorithm $\mathcal{A\_HEAP}$ (Figure 5.1) is a particular case of Algorithm $\mathcal{SSDST}$, in which the macro *execute*(v) is replaced by the macro *heap*(v) that sets $IV.v$ to the minimal value among itself and its children's IV-values.

Consider an abstract model, different from the shared-memory model, in which a node v, in order to have the heap property locally, can modify the variable $IV.J$ of some child J. Intuitively, since by executing Algorithm $\mathcal{SSDST}$, local mutual exclusion is satisfied in any configuration (see Property 1), a node can synchronize the swap of values with some child. We assume for now that the swap is done in an atomic step (macro *heap*), and we show in Section 5.2 how this is done in the shared-memory model.

Each node, besides the variable IV to be sorted, holds a variable $S \in \{A, B\}$, a pointer $i \in 0 \ldots |D_v| - 1$, and a variable $j \in \{-1, 0, \ldots, |D_v| - 1\}$ that either points to some child of node v that holds a value smaller than node v, or has the value -1 if either node v is a leaf or all its children have larger values. Thus, the total memory requirement of node v is $1 + 2\lceil \log(deg) \rceil$ bits per node (deg is the node degree).

For some node v, let $S = S.v$ and $IV = IV.v$. Predicate *check*(v, s) is defined in Section 3. If all children of v hold values greater than or equal to IV, then $min(v)$ returns the default value -1. Otherwise, $min(v)$ returns the index in the array D_v of a child of node v which holds the minimum value.

The guards $C1$-$C3$ "correct" the variable S of the node to some value in the set $\{A, B\}$ (a result of a fault or arbitrary initialization).

5.2 Heap Construction in the Shared-Memory Model

In Algorithm $\mathcal{HEAP}$ (Figure 5.2), each node v holds, besides the variable IV to be sorted, a variable $S \in \{A, B, X, Y\}$, a pointer i, a variable j, a variable J which is a pointer to some child, and a variable *tmpS* $\in \{A, B\}$. Variable *tmpS*

Algorithm 5.1. *S-S. Min-Heap in a Rooted Tree in the Abstract Model $\mathcal{A\_HEAP}$*

```
Macro heap(v) ::
  j = min(v)
  if (j ≥ 0) then J = D_v[j]; IV.v=IV.v+IV.J; IV.J=IV.v- IV.J; IV.v=IV.v-IV.J

Function min(v) ::
  if D_v = ⊥ then return -1
  else
      j = 0
      forall l ∈ {0, |D_v| − 1} do if (IV.D_v[j] > IV.D_v[l]) then j = l
      if (IV.D_v[j] < IV.v) then return j else return −1

Heap actions for any node v

ABB  S = B ∧ check(p_v, A) ∧ ∀i, 0 ≤ i < |D_v| : check(D_v[i], B)  ⟶  heap(v); S = A
BAA  S = A ∧ check(p_v, B) ∧ ∀i, 0 ≤ i < |D_v| : check(D_v[i], A)  ⟶  heap(v); S = B
```

stores the value of S temporarily while the swap is performed between node v and its child J. Thus, the total memory requirement of node v is $3+2\lceil\log(deg)\rceil$ bits per node (*deg* is the node degree).

For any node v, let $S = S.v$, $IV = IV.v$, $J = J.v$, $tmpS = tmpS.v$, $S_p = S.p_v$, $J_p = J.p_v$, $IV_p = IV.p_v$, $S_J = S.(J.v)$, and $IV_j = IV.(J.v)$. The macro $heap'(v, value)$ executes the first step of swapping between node v and the child $J = D_v[j]$, and the value *value* to be given to variable $S.v$ after the swap is performed is stored in variable *tmpS.v*.

Predicate *check*(v, s) has been defined in Section3. Function $min(v)$ is defined in Section 5.1.

In order to perform the swap, nodes v and J_v must change their S-value (from either A or B to either X or Y). Since node v will change its S-value after the swap, the value to-be for $S.v$ and the value of S_J are stored in variables *tmpS.v*, respectively *tmpS.J*, by each node. Node v changes its S-value to X (macro *heap*$'$) and node J changes its S-value to Y (Guard $S1$). The swap started by node v already in macro *heap*$'$ is continued by node J in Guard $S1$, and finished by node v in Guard $S2$ (where also node v restores its S). Once the swap is done, the S-values are restored back to A or B, node v in Guard $S2$, node J in Guard $S3$.

In Figure 4, nodes v and J swap their IV-values (a state of is a triple $S; IV; tmpS$).

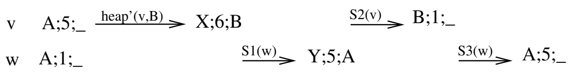


Fig. 4. Nodes v and J swap their IV values

Algorithm 5.2. *Self-stabilizing Heap in a Rooted Tree in the Shared-Memory Model $\mathcal{HEAP}$*

```
Macro heap'(v, tS) ::
  $j = min(v)$
  if $(j \geq 0)$ then $J = D_v[j]$; $tmpS.v = tS$; $IV.v = IV.v + IV.J$; $S.v = X$

Heap actions for any node $v$

ABB  $S = B \wedge check(p_v, A) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], B) \longrightarrow heap'(v, A)$
BAA  $S = A \wedge check(p_v, B) \wedge \forall i, 0 \leq i < |D_v| : check(D_v[i], A) \longrightarrow heap'(v, B)$

Synchronizing actions for any node $v$

S1   $S \in \{A, B\} \wedge p_v \neq \perp \wedge S_p = X \wedge J_p = v \longrightarrow IV{=}IV_p\text{-}IV$ ; $tmpS = S$ ; $S = Y$
S2   $S = X \wedge J \neq \perp \wedge S_J = Y \longrightarrow IV = IV - IV_J$ ; $S = tmpS$
S3   $S = Y \wedge p_v \neq \perp \wedge S_p \neq X \longrightarrow S = tmpS$
C1   $S = Y \wedge p_v = \perp \longrightarrow S = tmpS$
C2   $S = X \wedge D_v = \perp \longrightarrow S = tmpS$
C3   $S = X \wedge D_v \neq \perp \wedge \exists w \in D_v : S.w = X \longrightarrow S = tmpS$
```

5.3 Proof of Correctness of $\mathcal{A\_HEAP}$

The root node R has level 1. Besides local mutual exclusion, heap-building requires synchronization between neighboring nodes. Each node has a local clock measuring *pseudo-time* such that the comparison between the node and its child with the minimal IV value (and eventual swapping) is done when the two nodes have the same pseudo-time values.

For each configuration, the *pseudo-time* function Ψ is defined from the node to non-negative integers. Ψ is initially computed from the delay values, and is updated at each step.

Ψ_0, the pseudo-time at the initial configuration, is defined as follows:

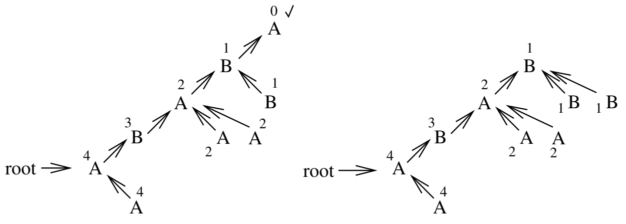
(i) given node v and its parent p_v, $\Psi_0(v) = \frac{d_0[v]+d_0[p_v]-1}{2}$, and
(ii) $\Psi_0(R) = max\{\Psi_0(v), v \in child_R\}$, where R is the root.

For example, given the configuration in Figure 3(c), the Ψ_0 values are given in Figure 5(a).

We observe that if a node v is enabled, then $\Psi_0(v) = \Psi_0(w)$ for all $w \in D_v$.

Definition 4. *Let Ψ_j and Ψ_{j+1} be the pseudo-time functions for two consecutive configurations in some execution $C_j \mapsto C_{j+1}$. Then Ψ_{j+1} is computed as follows:*

- if node v has executed during this step then $\Psi_j(v)$ and $\Psi_j(w)$ for all children $w \in D_v$ increase by 1: $\Psi_{j+1}(v) = \Psi_j(v) + 1$ and $\Psi_{j+1}(w) = \Psi_j(w) + 1$.
- if any child of the root R executes, $\Psi(R)$ is updated if necessary, i.e., $\Psi_{j+1}(R) = max_{w \in \text{child}_R}\{\Psi_{j+1}(w)\}$
- all other nodes u keep their current pseudo-time values, i.e., $\Psi_{j+1}(u) = \Psi_j(u)$.



(a) Start configuration (b) After one step

Fig. 5. Pseudo-time values

For example, given Ψ_0 from Figure 5(a), if the marked node executes, then the next pseudo-time values are the ones in Figure 5(b).

The following relations hold:

(i) $\Psi_0(R) \leq h$
(ii) $\Psi_0(v) \leq i + h - 1$ for $v \neq R$, where $i = \mathit{level}(v)$.

Thus $\Psi_0(v) \leq 2h - 1$, for any node v.

Let $\mathcal{E}(v, t)$ be the predicate: "Node v is enabled if $\Psi(v) = t$."

Observation 1. *(i) If $\mathcal{E}(v,t)$ is* true *then $\mathcal{E}(v,t+2k+1)$ is* false *and $\mathcal{E}(v,t+2k)$ is* true*, for all $k \geq 0$.*

(ii) If $\mathcal{E}(v,t)$ is false *and $t \geq \Psi_0(v)$ then $\mathcal{E}(v,t+2k+1)$ is* true *and $\mathcal{E}(v,t+2k)$ is* false*, for all $k \geq 0$.*

Property 4. Given a starting configuration C_0, and C_j some configuration after Algorithm $\mathcal{SSDST}$ has executed j steps, then the number of rounds elapsed is $q \leq min\{\forall \text{ nodes } v, \Psi_j(v)\}$.

Proof. A round has elapsed if all nodes enabled in the first configuration of the round have increased their Ψ values by at least one unit; thus the minimum value among all nodes has increased at least by one.

We assume that the values to be placed in min-heap order are distinct. (If necessary, we can add infinitesimal tie-breakers to the values.) Thus they can be arranged in a strict sorted order: $r_1 < r_2 < \ldots < r_n$, and we say that the value r_i has rank i.

Definition 5. *For any given configuration C of Algorithm $\mathcal{A\_HEAP}$, let l_i be the level of the node that holds the value r_i; we call the function $W(C) = \sum_{i=1}^{n} l_i i$ the* weighted path length *of the configuration C.*

The function W is strictly positive. It increases when a swap is executed between some node v that holds the value r_i and some child $w \in D_v$ that holds the value r_j, where $r_i < r_j$. The value by which W increases is $j - i$.

By Lemma 1, if the heap property does not hold at some node v, node v will execute a swap in finitely many rounds. Since $W(\mathcal{C})$ is an increasing integer function bounded by hn, it must converge in finitely many steps. Thus:

Observation 2. *Function W converges in finitely many of rounds. Let $\mathcal{C}_*$ be the configuration after convergence. Then $\mathcal{C}_*$ has the heap property.*

Let L_i be the level of the node that holds the value r_i in configuration $\mathcal{C}_*$. Array *pos* is defined as follows.

Definition 6. *Given j, $1 \leq j \leq n$, and some $t \geq 0$, the value* $\mathrm{pos}[j,t]$ *represents the level of node v that holds the value r_j when $\Psi(v) = t$.*

If initially, the element of r_i is held by the node v situated at level l_i and $\Psi(v) = t_0$, then we assume that for any t, $0 \leq t \leq t_0$, $pos[j,t] = pos[j,t_0]$.

First, we show that once the Ψ value of some node is t, the level $pos[j,t]$ of the element r_j is within a certain range (Property 5). In order to show that Algorithm $\mathcal{A\_HEAP}$ arranges the values as a heap, we show that after $7h/2-4$ rounds, $pos[j,t] = L_j$ for all j (Lemma 2).

Property 5. For any $t \geq 0$ and for any j, $1 \leq j \leq n$,

$$min\{L_j, Q[j,t]\} \leq pos[j,t] \leq max\{L_j, P[j,t]\}$$

where $P[j,t] = -t + 2L_j + 3h - 5$ and $Q[j,t] = t + 2L_j + 3 - 4h$, for any j and t.

Proof. Consider the predicates:

$\mathcal{P}(t)$: for any $j \in 1 \ldots n, pos[j,t] \leq max\{L_j, P[j,t]\}$
$\mathcal{Q}(t)$: for any $j \in 1 \ldots n, pos[j,t] \geq min\{L_j, Q[j,t]\}$

It can be shown by induction on t that $\mathcal{P}(t)$ holds. The proof that $\mathcal{Q}(t)$ holds is similar.

Lemma 2. *Algorithms $\mathcal{A\_HEAP}$ arranges the values into min-heap order in 7h/2-4 rounds; thus the stabilization time is $O(h)$ rounds.*

Proof. Follows from Property 5.

5.4 Reduction of Algorithm $\mathcal{HEAP}$ to $\mathcal{A\_HEAP}$

In this section we first show that Algorithm $\mathcal{HEAP}$ reduces to Algorithm $\mathcal{A\_HEAP}$. We can then conclude that, starting from an arbitrary configuration, in at most $4(7h/2-4)$ rounds, Algorithm $\mathcal{HEAP}$ arranges the values into min-heap order (Lemma 5).

Definition 7 (Reduction). *Given two different models of communication $\mathcal{M}$ and $\mathcal{M}'$, an algorithm $\mathcal{A}$ in the model $\mathcal{M}$ can be reduced to another algorithm $\mathcal{A}'$ in the model $\mathcal{M}'$ if there exists a one-to-many relation $\mathcal{R}$ from the set of system configurations in the model $\mathcal{M}$ to the set of the system configurations in the model $\mathcal{M}'$ such that the following conditions are* true*:*

i) *For each configuration of Algorithm $\mathcal{A}$ in the model $\mathcal{M}$ there exists at least one configuration of Algorithm $\mathcal{A}'$ in the model $\mathcal{M}'$.*
ii) *(Lifting property) Given $\mathcal{C}_1$ and $\mathcal{C}_2$ two configurations of Algorithm $\mathcal{A}$ in the model $\mathcal{M}$ such that $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ is an execution step of Algorithm $\mathcal{A}$, for any configuration $\mathcal{C}_1' \in \mathcal{R}(C_1)$, if Algorithm $\mathcal{A}'$ in the model $\mathcal{M}'$ starts in $\mathcal{C}_1'$ there exists at least one execution path that starts in $\mathcal{C}_1'$ and ends in some configuration $\mathcal{C}_2' \in \mathcal{R}(\mathcal{C}_2)$.*

If $\mathcal{A}$ accomplishes a task in the model $\mathcal{M}$ and $\mathcal{A}$ reduces to $\mathcal{A}'$, then by Definition 7, $\mathcal{A}'$ accomplishes the same task in the model $\mathcal{M}'$.

We now show that Algorithm $\mathcal{HEAP}$ reduces to Algorithm $\mathcal{A\_HEAP}$. Let $S_v = (s_v, x_v, J_v)$ be the set of all variables of node v in order (S, IV, J) used by Algorithm $\mathcal{A\_HEAP}$ in the abstract model. Let $S_v^{t_v} = (s_v, x_v, t_v, J_v)$ be the set of all variables of node v in order $(S, IV, tmpS, J)$ used by Algorithm $\mathcal{HEAP}$ in the shared-memory model.

Then $\mathcal{R}$ is defined as follows:

$$\mathcal{R}(S_1, \ldots S_n) = \{(S_1^{t_1}, \ldots S_n^{t_n}), t_i \in \{A, B\}, \forall i, 1 \le i \le n\}$$

For each state S_i of some configuration $\mathcal{C}_1$ of Algorithm $\mathcal{A\_HEAP}$ in the abstract model, $1 \le i \le n$, there exists two possible states S_i^A and S_i^B in the shared-memory model. Thus for each configuration $\mathcal{C}_1$ there exists 2^n configurations in $\mathcal{R}(\mathcal{C}_1)$ of Algorithm $\mathcal{HEAP}$ in the shared-memory model, thus Condition (i) of Definition 7 is satisfied. We are left to show that Condition (ii) of Definition 7 is satisfied (Lemma 3).

Lemma 3. *Given $\mathcal{C}_1$ and $\mathcal{C}_2$, two configurations of Algorithm $\mathcal{A\_HEAP}$ in the abstract model, such that $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ is an execution step of Algorithm $\mathcal{A\_HEAP}$; for any configuration $\mathcal{C}_1' \in \mathcal{R}(\mathcal{C}_1)$, if Algorithm $\mathcal{HEAP}$ in the shared-memory model starts in $\mathcal{C}_1'$ there exists at least one execution path that starts in $\mathcal{C}_1'$ and ends in some configuration $\mathcal{C}_2' \in \mathcal{R}(\mathcal{C}_2)$.*

Proof. We give a sketch of the proof. A node state contains all the variables stored at that node. The system configuration contains the states of all the nodes. An execution step is a transition from one configuration to another. We break the system configuration into a number of *chunks*. A *chunk* is a set of a node and its descendants in the tree such that the first node in each chunk is enabled, and all the descendants of the first node reachable by a path of disabled nodes are added to the chunk. We build the set of chunks starting from the root in depth-first-search (DFS) order. If the root node is currently disabled, then the root and all nodes reachable from the root reachable by a path of disabled nodes are not part of any chunk. We call the set of those nodes the *null chunk*.

Given a configuration, there is a unique way to break it into chunks. An execution step of Algorithm $\mathcal{A\_HEAP}$ in the abstract model in one chunk affects only the nodes' states in that chunk.

From Property 1 we know that if a non-leaf node is enabled, its children are disabled. So, except for the leaf nodes, every chunk contains at least two nodes.

If the chunk contains at least two nodes, then the last node in the chunk is disabled, so it cannot affect the state of the first node of other chunks.

Instead of considering an execution step between *global* configurations, we consider an execution step between the chunks of a global configuration.

If the starting state of the node is either A or B, then the value to be sorted is its initial value. If some node starting state is either X or Y, then it is possible for some of the three steps of the swap to be applied (see Section 5) and the initial value of that node to be modified accordingly, and that modified value to be sorted. This drawback is caused by arbitrary initialization, and would be encountered even if we had used an extra variable for swapping.

We recall that node $J.v$ is the child of node v that holds the minimal IV value among all node v's children. The variable $J.v$ is $\perp$ if and only if node v is a leaf node ($child.v = \perp$).

For any node v such that $S_v = X$, either S_v remains X and then the node $J.v$ will have its S equal to Y in at most three rounds (by executing Action $S1$), or v changes its S to A or B in at most one round.

For any node v such that $S.v = X \wedge S.(J.v) = Y$ then $IV.v$ gets the value $IV.(J.v)$ and then node v changes its S_v to A or B in at most one round. Node $J.v$ had already stored in $IV.(J.v)$ the old value of $IV.v$ (by executing Action $S1$) and will restore its $S_{J.v}$ from Y to either A or B (depending on the value of *tmpS*) in at most one round. We can then conclude that if $S.v$ is either X or Y, then in at most four rounds $S.v$ is either A or B (Lemma 4).

Lemma 4. *For any node v, if $S.v \in \{X, Y\}$, in at most four rounds $S.v$ becomes either A or B.*

Lemma 5. *Starting from an arbitrary configuration, in at most* $4(7h/2 - 4)$ *rounds, Algorithm $\mathcal{HEAP}$ arranges the n values in min-heap order.*

Proof. From Lemma 4, each swap takes at most 4 rounds. From Lemma 2, if a swap takes at most 1 round, then heapification takes at most $7h/2 - 4$ rounds. Since the swap takes at most 4 rounds, we obtain a total of at most $14h - 16$ rounds.

6 Conclusion

In this paper, we present the first self-stabilizing algorithm for simultaneously activating non-adjacent processes in a rooted tree, called $\mathcal{SSDST}$. The algorithm is optimal in the space complexity, and asymptotically optimal in the time complexity. We then give two applications of the proposed algorithm for rooted trees, a time and space optimal solution to the local mutual exclusion problem (Algorithm $\mathcal{LMET}$) and a space and (asymptotically) time optimal solution to the min-heap problem (Algorithm $\mathcal{HEAP}$).

All algorithms are self-stabilizing and uniform, and they work under the unfair distributed daemon.

In proving the time complexity of heap-building, we use the notion of *pseudo-time*. Pseudo-time is similar to *logical time* introduced by Lamport [12].

We expect that Algorithm $\mathcal{SSDST}$ can be used to obtain optimal space solutions for other problems in a rooted tree. For example, for broadcasting m messages, a solution based on Algorithm $\mathcal{SSDST}$ stabilizes in at most $2h + 2m - 5$ rounds (the root node executes m times).

References

1. L. Alima. Self-stabilizing max-heap. *Proceedings of the ICDCS Workshop on Self-stabilizing Systems*, pages 94–101, 1999.
2. L. Alima, J. Beauquier, A. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. *Proceedings of the 18-th ICDCS*, pages 102–109, 1998.
3. A. Arora and M. Nesterenko. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62:766–791, 2002.
4. D. Bein, A. Datta, and V. Villain. Snap-stabilizing optimal binary-search-tree. *Proceedings of the 7-th International Symposium on Self-Stabilizing Systems*, 2005.
5. B. Bourgon and A. Datta. A self-stabilizing distributed heap maintenance protocol. *Proceedings of the Second Workshop on Self-stabilizing Systems*, 1995.
6. A. Bui, A. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 78–85. IEEE Computer Society, 1999.
7. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (second edition).* MIT Press, 2001.
8. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
9. 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.
10. T. Herman and T. Masuzawa. Available stabilizing heaps. *Information Processing Letters*, 77:115–121, 2001.
11. C. Johnen, L. Alima, A. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3 & 4):327–340, 2002.
12. L. Lamport. Time, clocks and the ordering of events in a distributed systems. *Communications of the ACM*, 21:558–565, 1978.
13. S. Ukena, M. Hasegawa, Y. Katayama, T. Masuzawa, and H. Fujiwara. A self-stabilizing max-heap protocol in tree networks. *Electronics and Communications in Japan, Part III: Fundamental Electronic Science (English translation of Denshi Tsushin Gakkai Ronbunshi)*, 86(9):63–72, 2003.