Scientific Annals in Computer Science vol. 18, 2008
"Alexandru Ioan Cuza" University of Iaşi, Romania

Synchronization Algorithms on Oriented Chains

Doina Bein,[1] Ajoy K. Datta,[2] Lawrence L. Larmore[3]

Abstract

We present a space- and time-optimal self-stabilizing algorithm, $\mathcal{SSDS}$, for a given synchronization problem on asynchronous oriented chains. $\mathcal{SSDS}$ is uniform and works under the unfair distributed daemon. From $\mathcal{SSDS}$ we derive solutions for the local mutual exclusion and distributed sorting. Algorithm $\mathcal{SSDS}$ can also be used to obtain optimal space solutions for other problems such as broadcasting, leader election, and mutual exclusion.

1 Motivation and Background

Distributed algorithms aim to reduce the amount of communication among the nodes in the system (the number of messages exchanged for message-passing or the number of memory accesses – reading or writing into variables – for shared-memory model), while keeping a relative low memory requirement per node and a low time complexity. Fault-tolerance is the ability of an algorithm to withstand transient faults (still perform its function), where by a fault we mean that the variables of some nodes are changed to some arbitrary values by some external action. Self-stabilization is the strongest case of fault-tolerance: An algorithm is *self-stabilizing* when, "regardless of

[1]Contact author. Applied Research Laboratory, The Pennsylvania State University, PA 16802, USA. Email: siona@psu.edu.

[2]School of Computer Science, University of Nevada, Las Vegas, NV 89154, USA. Email: datta@cs.unlv.edu.

[3]School of Computer Science, University of Nevada, Las Vegas, NV 89154, USA. Email: larmore@cs.unlv.edu.

its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." (Dijkstra [1]) An algorithm which is not self-stabilizing may stay in a illegitimate state forever and never be able to perform its function. Self-stabilizing distributed algorithms aim to achieve performance comparable to non-stabilizing distributed algorithms in the presence of transient faults or an arbitrary initialization. Local mutual exclusion is a special case of the synchronization of nodes' execution in which no two neighboring nodes can execute simultaneously. Given n arbitrary values, not necessarily distinct, and n-node oriented graph (e.g., oriented chain, rooted tree), each node holding a single value, sorting distributes the values among the nodes in the increasing order of the node topological position. For example, in an oriented chain network, the order is either from left to right or from right to left; for a rooted tree network, the order is either from the root to the leaves or from the leaves to the root. In this paper we consider sorting in a asynchronous oriented chain network. Given n values and n nodes arranged as a chain, sorting distributes the values among nodes in the increasing order from the leftmost to the rightmost node.

In this paper, we propose a general synchronization scheme and space optimal solutions for the local mutual exclusion problem and distributed sorting on asynchronous oriented chains. In order to compute the time complexity of any algorithm running on an asynchronous system, we use the definition of a *round* [2]. A round is a minimal sequence of computation steps such that each process that was enabled in the first configuration of the sequence has executed at least once during the sequence.

Related Work. Non fault-tolerant distributed sorting was studied in [3, 4, 5]. Flocchini et al. [3] analyze the relationship between sorting and election in an anonymous asynchronous ring. Sasaki [4, 5] gives time optimal ($n-1$ steps) solutions to distributed sorting in a synchronous [4] and asynchronous oriented chain [5]. The space complexity per node is $O(L)$, where L is the maximal size of the initial values. Initially, at each node copies of values are created and these copies are used for sorting. Both algorithms are non fault-tolerant: if a fault changes the values to the copy to some arbitrary value, the outputs of both algorithms are incorrect – the initial values are not sorted, but the erroneous ones instead.

Fault-tolerant algorithms for synchronization and sorting are given in [6, 7]. A general fault-tolerant scheme for solving any synchronization problem whose safety specification can be defined using a local property is given in [6]. A stabilizing min-heap algorithm that uses $O(h)$ rounds and $O(\log L)$

bits is given in [7], where h is the height of the heap. The drawback of this algorithm is that it assumes a stronger model in which a node can read and write into all its neighbors' variables.

Contributions. We consider the following synchronization problem: "Given an asynchronous system and a positive integer $t > 0$, find the minimum positive integer T such that every node is enabled at least t times within T rounds, and whenever some node is enabled, all of its direct neighbors are disabled." Our contribution is threefold.

We propose a general self-stabilizing algorithm, $\mathcal{SSDS}$ (Figure 1), that solves this problem for an asynchronous oriented chain in minimum number of rounds. $\mathcal{SSDS}$ is optimal in space complexity, *i.e.*, it uses one bit space in every node, and is asymptotically optimal in time complexity, *i.e.*, within $n-2+2t$ rounds every node will be enabled at least t times. For a synchronous oriented chain, after $n-1$ steps, every node is enabled every other round.

We use $\mathcal{SSDS}$ to derive an algorithm for the local mutual exclusion problem on asynchronous oriented chains, $\mathcal{LME}$ (Figure 5), that satisfies the strong safety property, *i.e.*, in any configuration, there exists at least one privileged node; and distributed sorting in non-decreasing order from left to right. $\mathcal{LME}$ uses two bits per node and stabilizes in 0 rounds (it is snap-stabilizing [8, 9]). We also use $\mathcal{SSDS}$ to derive two algorithms for distributed sorting on asynchronous oriented chains. Algorithm $\mathcal{S}_1$ (Figure 7) is space optimal and time asymptotically optimal; it sorts the n values within $4(2n-2)$ rounds and uses a total of three bits per node, thus an improvement over the algorithms of [4, 5]. Algorithm $\mathcal{S}_2$ (Figure 9) is almost time optimal; it sorts the n values within $2n-1$ rounds (the minimum number of rounds required is $2n-2$) and uses $L+1$ bits per node, where L is the maximum size of the initial values in the chain.

Outline of the paper. Section 2 contains some preliminary notions and the synchronization problem. Algorithm $\mathcal{SSDS}$ is given in Section 3. Algorithm $\mathcal{LME}$ is in Section 4. In Section 5 we start with a sorting algorithm in the EREW[4] (exclusive read, exclusive write) model, Algorithm $\mathcal{A}\_\mathcal{S}_1$ (Figure 6). Using the shared memory model of communication we then present a sorting algorithm that uses a constant number of bits (Algorithm $\mathcal{S}_1$, Figure 7) and almost minimum number of rounds (Algorithm $\mathcal{S}_2$, Figure 9). In Section 6 we show why Algorithm $\mathcal{SSDS}$ cannot work under the read/write atomicity model without increasing the memory requirement per

[4]In this model, any process can access any register, but no two processes can access the same register in the same step.

node. We finish with concluding remarks in Section 7 and acknowledgments.

2 Preliminaries

We consider an asynchronous bi-directional, oriented chain of n nodes[5]. Each node v has a finite set of variables of various types and can distinguish between its left (l_v) and its right (r_v) neighbor. The left-right orientation is consistent among all nodes in the network. Let L and R be the leftmost, respectively rightmost node. If a node v has only one neighbor, the value of the missing neighbor is represented as $\perp$.

Nodes communicate with each other by shared memory: A node can read and write into its own variables, but can only read the variables of its neighbors. The *state* of a node is the set of the values of its variables. A *system state* (or *configuration*) is simply a choice of a state for each node. An *execution* is a finite or infinite sequence of configurations, of maximum length. For any configuration c, let E_c be the set of all executions that start from configuration c, *i.e.* the starting configuration of the algorithm is c.

A node executes an algorithm that is a finite set of guarded actions of the form: $\langle label \rangle :: \langle guard \rangle \rightarrow \langle action \rangle$. If all the nodes execute the same algorithm, then the algorithm is called *uniform*. The *guard* is a Boolean expression that involves the node's variables and possibly the variables of its direct neighbors. The *action* is a finite set of statements that involve only the node's variables. If an action has its guard evaluated to *true*, then it is called *enabled*. A node with at least one enabled action is called *enabled node*. If all the guards of the algorithm are mutually exclusive, then the algorithm is called *deterministic*. At the beginning of a computation step, a *distributed daemon* selects a non-empty subset of enabled nodes to execute; at the end of the computation step, all selected nodes have executed. The selected nodes are called *privileged*. Each privileged node executes exactly one of its enabled actions. The guard evaluation and the execution of the corresponding action are considered to be done in one atomic step. There are several types of distributed daemons, but the most common are the weakly fair and the unfair. With a *weakly fair* daemon, a continuously enabled node will eventually become privileged after a finite number of rounds. In this paper we consider the *unfair* daemon: a continuously enabled process may not be selected for execution unless it becomes the only enabled process. The unfair daemon is the strongest type of distributed daemon: a self-stabilizing

[5]We use the terms of node and process interchangeably.

algorithm written under the unfair daemon will be self-stabilizing also under the weakly fair daemon, but not vice-versa.

Definition 1 (Synchronization Problem) *Given a system and a positive integer $k > 0$, find the minimum positive integer T such that every node in the system is enabled to execute a given task at least k times within T rounds, and whenever a node is enabled, neither neighboring node is enabled.*

Given $\mathcal{C}$, the set of all possible states, and a predicate $\mathcal{P}$ defined over a state in $\mathcal{C}$, the predicate $\mathcal{P}$ is evaluated to either $true$ or $false$ for each state in $\mathcal{C}$. Thus $\mathcal{C}$ is partitioned into two sets: one set that contains all the states for which $\mathcal{P}$ is $true$, which we denote by $\mathcal{L}_{\mathcal{P}}$, and one set that contains all the states for which $\mathcal{P}$ is $false$. The set $\mathcal{L}_{\mathcal{P}}$ is called the set of all *legitimate states with respect to* $\mathcal{P}$, or the set of all *legitimate states* when $\mathcal{P}$ is understood. The states not in $\mathcal{L}_{\mathcal{P}}$ are called *illegitimate*. For various predicates $\mathcal{P}$, the set of legitimate states varies.

We present the notion of self-stabilization from [10], based on the notion of *closed attractor* [10]. Let C_1 and C_2 be subsets of C. C_2 is a *closed attractor* for C_1 if and only if both conditions are true: (i) for any initial state c_1 in C_1 and for any execution e starting from c_1, $e = c_1, c_2, ..$, there exists $i \geq 1$ such that for any $j \geq i$, $c_j \in C_2$, and (ii) any execution starting from a configuration in C_2 reaches a configuration in C_2.

From [10] we have: Given a predicate $\mathcal{P}$, an algorithm S is called *self-stabilizing with respect to* $\mathcal{P}$ (or simply, *self-stabilizing*, when $\mathcal{P}$ is understood) if and only if $\mathcal{L}_{\mathcal{P}}$ is a closed attractor for $\mathcal{C}$.

3 Algorithm $\mathcal{SSDS}$

Algorithm $\mathcal{SSDS}$ solves the synchronization problem presented in Definition 1 for an oriented chain and some generic macro called *execute*. Macro $execute(v)$ is an application specific task; e.g., accessing the critical section for local mutual exclusion, local sorting for distributed sorting. Each node holds a variable $S \in \{A, B\}$, thus needs only 1 bit. For any node v, let S, S_l, and S_r represent the S-value of node v, l_v, and r_v, respectively. Predicate $check(v, s)$ checks if node v is exists, and if so, whether its S-value is s.

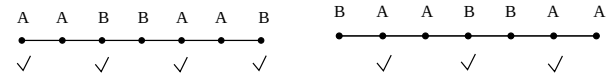
The guarded actions ABB and BAA are enabled at node v when the following conditions are *true*: (i) either v has no left neighbor or the left neighbor's S-value is different from its S-value, and (ii) either v has no right neighbor or the right neighbor's S-value is the same as its S-value.

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

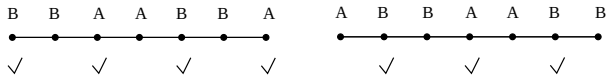
Actions for any node v

ABB	$S = B \wedge check(l_v, A) \wedge check(r_v, B)$	$\longrightarrow$	$execute(v)$; $S = A$
BAA	$S = A \wedge check(l_v, B) \wedge check(r_v, A)$	$\longrightarrow$	$execute(v)$; $S = B$

Figure 1: *Algorithm* $\mathcal{SSDS}$



(a) Starting configuration (b) After one step



(c) After two steps (d) After three steps

Figure 2: Four steps in 7-node chain

Consider a 7-node oriented chain with the starting configuration shown in Figure 2(a). We assume that all enabled nodes are selected by the daemon to execute in the same computation step; this is called a *synchronous* execution. In Figure 2(a), the odd-numbered nodes are enabled (from left to right the first, third, fifth, and seventh). At the end of one computation step, the configuration reached is in Figure 2(b); the even-numbered nodes are enabled. After the second computation step, the configuration reached is in Figure 2(c); the odd-numbered nodes are enabled. After the third computation step, the configuration reached is in Figure 2(d); the even-numbered nodes are enabled. After the fourth computation step, the system reaches the configuration in Figure 2(a); this cycle repeats forever.

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

Algorithm $\mathcal{SSDS}$ stabilizes within $n - 2 + 2k$ rounds to the predicate k-$Exec \equiv \{\forall k > 0\ \exists T$ such that $\forall$ node $v : live(v, k, T) \wedge safe(v)\}$ where $live(v, k, T) \equiv \{v$ executes macro $execute$ at least k times within T rounds$\}$ and $safe(v) \equiv \{$ when v is enabled, no neighbor is enabled $\}$.

Th guards of Algorithm $\mathcal{SSDS}$ guarantee the following propositions:

- **Local mutual exclusion** Whenever a node is enabled, no neighbors are enabled (Proposition 5).

- **No deadlock** There is always an enabled node (Proposition 6).
- **Fairness** After a node executes, it becomes disabled and remains so until all its neighbors execute (Proposition 7).
- **No starvation** - During the first $n-2+2k$ rounds, every node is enabled at least k times (Corollary 12 of Lemma 11). Thus $T = n-2+2k$ is the minimum number of rounds.

$\mathcal{SSDS}$ works under the unfair distributed daemon (Proposition 14).

If $n = 1$ and the node's $S = A$, then the node alternately executes Actions BAA and ABB forever. We will assume $n > 1$ henceforth.

A *normal starting configuration* is the n-length prefix of $(AABB)^n$, which is a $4n$-length string obtained by concatenating n copies of $AABB$. We note that in a normal starting configuration, the odd-numbered nodes are enabled (see Figure 2(a) for particular case $n = 7$). If the system has a synchronous execution, a normal starting configuration is reachable from any configuration within $n-1$ steps.

Any configuration of length n can be mapped to a unique binary $(n-1)$-bit string, called *difference-string*.

Definition 2 *Given an n-length configuration, $C = S_1S_2 \ldots S_n$, a* difference-string *is the $(n-1)$-length binary string $DS_C = b_1b_2 \ldots b_{n-1}$ such that $b_i = 0$ if $S_i = S_{i+1}$, 1 otherwise.*

A given difference-string corresponds to two configurations. For example, the difference-string 101010100 corresponds to either configuration $ABBAABBAAA$ or $BAABBAABBB$.

Remark 3 *Given a difference-string DS and a value S of some node, the corresponding configuration C is uniquely defined.*

Given a node v that is the i^{th} node from the left ($1 \leq i \leq n$) and a configuration C, let

$$DS_C(v) = \begin{cases} b_{i-1}b_i & \text{if } 2 \leq i \leq n-1 \\ b_0 & \text{if } i = 1 \\ b_{n-1} & \text{if } i = n \end{cases}$$

From the code of Algorithm $\mathcal{SSDS}$, we observe that:

Observation 4 *Given any configuration C:*
(i) Action ABB or BAA is enabled at the leftmost node L if and only if $DS_C(L) = 0$ and the execution of the guard changes $DS_C(L)$ from 0 to 1.
(ii) Action ABB or BAA is enabled at the rightmost node R if and only if $DS_C(v) = 1$ and the execution of the guard changes $DS_C(v)$ from 1 to 0.
(iii) For $n > 2$, Guarded action ABB or BAA is enabled at some node v other than L and R if and only if $DS_C(v) = 10$ and the execution of either guard changes $DS_C(v)$ from 10 to 01.

Proposition 5 *For any configuration C and node v, if node v is enabled to execute then neither node l_v nor r_v (if they exist) is enabled.*

Proof. Assume node v is enabled. Based on its position in the chain we have three cases:

1) v is the leftmost node L. If v is enabled, from Observation $4(i)$ we have $DS_C(v) = 0$; thus $DS_C(r_v)$ starts with a 0. From Observation $4(ii, iii)$, node r_v is enabled if $DS_C(r_v) \in \{10, 1\}$. Thus $DS_C(r_v)$ should start with a 1 for the node r_v to be enabled. Contradiction.

2) v is the rightmost node R. If v is enabled, from Observation $4(ii)$ we have $DS_C(v) = 1$; thus $DS_C(l_v)$ ends with a 1. From Observation $4(i, iii)$, node l_v is enabled if $DS_C(l_v) \in 0, 10$. Thus $DS_C(r_v)$ should end with a 0 for the node r_v to be enabled. Contradiction.

3) v is a node other than L or R. If v is enabled, from Observation $4(iii)$ we have $DS_C(v) = 10$; thus $DS_C(l_v)$ ends with a 1 and $DS_C(r_v)$ starts with a 0. From Observation $4(i, iii)$, node l_v is enabled if $DS_C(l_v) \in \{0, 10\}$. Thus $DS_C(l_v)$ should end with a 0; contradiction. From Observation $4(ii, iii)$, node r_v is enabled if $DS_C(r_v) \in \{10, 1\}$. Thus $DS_C(r_v)$ should start with a 1; contradiction. □

Proposition 6 *In any configuration C there exists at least one enabled node.*

Proof. Since $DS_C \in (0 + 1)^n$, we have three cases:

1) If DS_C starts with 0 then by Observation $4(i)$ the leftmost node L is enabled to execute.

2) If DS_C starts with a 1 and contains the substring 10 then, by Observation $4(iii)$, some node v is enabled to execute.

3) If DS_C is 1^n (starts with a 1 and does not contain the substring 10). Then DS_C ends with 1 and, by Observation $4(ii)$ the rightmost node R is enabled to execute. □

Proposition 7 *For any node v, if node v is enabled and is selected by the daemon to execute, after it executes all v's actions are disabled.*

Proof. From Observation $4(i, ii)$, if node v is either L or R, after it executes an enabled guard, it becomes disabled. From Observation $4(iii)$, a node v other than L or R is enabled if $DS_C(v)$ is 10. But once node v executes and configuration C changes to configuration C', then $DS_{C'}(v)$ is 01, so node v is disabled. □

In showing that by executing Algorithm $\mathcal{SSDS}$ on an oriented chain, after $n-1+2t$ rounds, every node is enabled at least t times, we need some additional notations, definitions and propositions.

Given two nodes v and r_v with $S.v = a$ and $S.r_v = b$, by notation "$a \leftarrow b$" we denote that state b does not block state a from being enabled (in order for v being in state a to be enabled, $S.r_v$ has to be b). The notation $a \rightarrow b$ denotes that state a does not block state b from being enabled (in order for r_v being in state b to be enabled, $S.v$ has to be a).

For example the guard of Action BAA can be re-written as $B \rightarrow A \leftarrow A$, and the guard of Action ABB can be re-written as $A \rightarrow B \leftarrow B$.

We can use the above notation to define layers as follows. We start by fixing the layer of node L, then proceed the right of the chain. If node v is on a certain layer and $S.v \rightarrow S.r_v$, then r_v is one layer higher. If $S.v \leftarrow S.r_v$ then r_v is one layer lower. A configuration has a sawtooth-like level ordering, where the peak nodes are the enabled ones.

The difference-string of a given configuration is consistent with the orientations of the arrows between consecutive S values (1 for ↗ and 0 for ↖). For example, for the configuration $BAAABBABABBBABAA$ of a 16-node network, the levels are drawn in Figure 3.

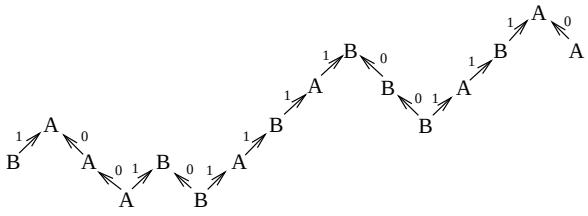


Figure 3: Configuration $BAAABBABABBBABAA$

Definition 8 (Node Delay) *Given a configuration, for each node v we define delay$[v]$ to be an integer between 0 and $n-1$, calculated recursively*

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* $delay[v] = delay[u] + 1$*. If* $S.v \leftarrow S.u$ *then* $delay[v] = delay[u] - 1$*.*

The delay value of a node is the maximum number of rounds a node must wait until it becomes enabled. The delay values of the nodes in Figure 3 are given in Figure 4.

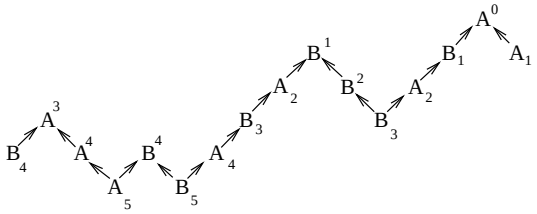


Figure 4: Delay values for configuration $BAAABBABABBBABAA$

After a node executes, the arrows (or the arrow, for the chain extremities) to that node are reversed. The delay values are then recalculated.

Proposition 9 *In any configuration, if* w *is a neighbor of* v *then* $delay[w] = delay[v] \pm 1$*.*

Proof. From Definition 8. □

Proposition 10 *For any* $t > 0$*:*
(i) If $S.v \rightarrow S.r_v$ *then node* v *cannot execute its enabled guard for the* t^{th} *time until* r_v *has executed its enabled guard for the* t^{th} *time, and node* r_v *cannot execute its enabled guard for the* $(t+1)^{\text{st}}$ *time until node* v *has executed its enabled guard for the* t^{th} *time.*
(ii) If $S.v \leftarrow S.r_v$ *then node* r_v *cannot execute its enabled guard for the* t^{th} *time until node* v *has executed its enabled guard for the* t^{th} *time, and node* v *cannot execute its enabled guard for the* $(t+1)^{\text{st}}$ *time until node* r_v *has executed its enabled guard for the* t^{th} *time.*

Proof. In case *i)*, in order for node v to be enabled, node r_v must change the orientation of the arrow between itself and node v. This will occur after node r_v is enabled. By Proposition 7, after node r_v executes its guard for the t^{th} time, it becomes disabled. Then for node r_v to become enabled again and to execute its enabled guard for the $(t+1)^{\text{st}}$ time, node v has to execute (for the t^{th} time) and change the orientation of the arc toward node r_v.

Case *ii)* is similar. □

Let d_0 be the array of the delay values in the starting configuration, and D_0 be the maximum value of d_0. By the definition of array *delay*, $1 \leq D_0 \leq n-1$.

Lemma 11 *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 $q \geq d_0[v] + 2t - 1$.

We prove by induction on $q \geq 1$ that Predicate $\mathcal{P}(q)$ holds.

Basic step $q = 1$. If $q = 1$, this implies that $d_0[v] = 0$ and $t = 1$. Since $d_0[v] = 0$, node v is currently enabled for the first time and it will execute within one round.

Inductive step for $q > 1$, $\mathcal{P}(q-1)$ holds. We have that $q \geq d_0[v]+2t-1$, and we must show that node v executes t times within the first q rounds.

From the induction hypothesis, we have that node v has executed $t-1$ times within the first $d_0[v] + 2t - 3$ rounds.

Let u be the left neighbor of node v. (The proof for the right neighbor is similar.) From Proposition 9, $d_0[u] = d_0[v] \pm 1$. Thus we have two cases:

1) $d_0[u] = d_0[v] - 1$. Since $q \geq d_0[v] + 2t - 1$, this implies that $q - 1 \geq d_0[v] - 1 + 2t - 1$, and further $q - 1 \geq d_0[u] + 2t - 1$. From the induction hypothesis, $\mathcal{P}(q-1)$ holds for every node, including node u. Thus node u executes t times within $q - 1$ rounds. From Proposition 10, node u does not block node v from being enabled for the t^{th} time during round q.

2) $d_0[u] = d_0[v] + 1$. Since $q \geq d_0[v] + 2t - 1$, this implies that $q - 1 \geq d_0[v] + 1 + 2t - 3$, and further $q - 1 \geq d_0[u] + 2(t-1) - 1$. From the induction hypothesis, $\mathcal{P}(q-1)$ holds for every node, including node u. Thus node u executes $t - 1$ times within $q - 1$ rounds. From Proposition 10, node u does not block node v from being enabled for the t^{th} time during round q.

Neither the left neighbor of v nor the right neighbor of v blocks node v from being enabled for the t^{th} time at the beginning of round q. Thus, node v is enabled at the beginning of round q and it will execute for the t^{th} time by the end of the round. □

Corollary 12 *For any node v and any value $t > 0$, during the first $n-2+2k$ rounds every node is enabled at least k times.*

Proof. A node executes only if it is currently enabled. From Lemma 11 and the definition of a round, if node node v executes t times within the first $d_0[v] + 2t - 1$ rounds, it has to be enabled t times within the first $d_0[v] + 2t - 2$ rounds. □

Algorithm $\mathcal{SSDS}$ 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 eventually becomes the only enabled node. If a node v is enabled but not selected by the distributed daemon, it remains enabled (Proposition 13). 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 (Proposition 14).

Proposition 13 *If a node v is enabled to execute but is not selected by the daemon, it remains enabled until it will be selected.*

Proof. If some node v is enabled, neither of the existing neighbors is enabled (Proposition 5). The neighboring nodes remain disabled until v executes. □

Proposition 14 *Every continuously enabled node will be eventually selected by the unfair distributed daemon after a finite number of rounds.*

Proof. By contradiction. Assume that there exists a continuously enabled node v, but the unfair daemon never selects it for execution. Since an execution of Algorithm $\mathcal{SSDS}$ is infinite, starting from any arbitrary state, then there exists at least one node u, $u \neq v$ such that u is executed infinitely often.

If node u executes infinitely often, then both neighbors of u execute infinitely often. The reason is as follows: After u executes, it becomes disabled (Proposition 7); the arrows (or the arrow, for the chain extremities) to u are reversed. Node u becomes enabled again only after its neighbors execute and reverse again the arrows.

Let A be the maximal set of nodes in the chain that execute infinitely often. Thus, if $u \in A$, then $left(u), right(u) \in A$. Recursively, the left and the right neighbors of $left(u)$ and $right(u)$ are also in A. Thus A consists of all nodes. By our assumption, $v \notin A$, which is a contradiction to the statement that A consists of all nodes. □

4 Self-Stabilizing Local Mutual Exclusion Algorithm $\mathcal{LME}$

Each node holds a variable S that takes values in the set $\{A, B\}$, and a Boolean variable *request* that is *true* whenever the process requests access to its critical section CS. For a node v, let $S = S.v$ and $request = request.v$. Predicate $check(v, l)$ has been defined in Section 3.

Actions for any node v

ABB	$S = B \wedge check(l_v, A) \wedge check(r_v, B)$	$\longrightarrow$	if $request$ then CS; $request = false$ $S = A$
BAA	$S = A \wedge check(l_v, B) \wedge check(r_v, A)$	$\longrightarrow$	if $request$ then CS; $request = false$ $S = B$

Figure 5: *Algorithm* $\mathcal{LME}$

A protocol solves the local mutual exclusion problem if any configuration of the system running the protocol has two properties ([11]): *(i) safety* - no two neighboring nodes have guarded commands that execute the critical section (CS) enabled, and *(ii) liveness* - a node requesting to execute its CS will eventually do so.

Proposition 5 shows that Algorithm $\mathcal{LME}$ has the safety property. Proposition 6 shows that Algorithm $\mathcal{LME}$ has the liveness property.

5 Self-Stabilizing Distributed Sorting Algorithms

In this section we present three algorithms for the distributed sorting problem in an oriented chain: $\mathcal{A}\_\mathcal{S}_1$ (in Section 5.1), $\mathcal{S}_1$ (in Section 5.2), and $\mathcal{S}_2$ (in Section 5.3). Algorithm $\mathcal{A}\_\mathcal{S}_1$ is implemented in the EREW model. Algorithms $\mathcal{S}_1$ and $\mathcal{S}_2$ are 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: $x = x + y$, $y = x - y$, and $x = x - y$.

5.1 Distributed Sorting on an Oriented Chain

Each node, besides the variable S, holds one variable IV to be sorted. Algorithm $\mathcal{A}\_\mathcal{S}_1$ (Figure 6) is a particular case of Algorithm $\mathcal{SSDS}$, in which

the macro $execute(v)$ is replaced by the macro $swap(v, r_v)$ that swaps the values $IV.v$ and $IV.r_v$.

In the EREW model of communication, in order to execute the swap a node v modifies the IV variables of its right neighbor r_v.

Intuitively, since by executing Algorithm $\mathcal{SSDS}$, local mutual exclusion is satisfied in any configuration, a node can synchronize the swap with its right neighbor. We assume for now that the swap is done in one atomic step (macro *swap*), and we show in Sections 5.2 and 5.3 how the swap is actually done in the shared memory model.

For a node v, let $S = S.v$, $S_l = S.l_v$, $S_r = S.r_v$. Predicate $check(v)$ has been defined in Section 3.

Macro $swap(v, w)$ ::
if $(w \neq \perp \wedge IV.v > IV.w)$ then $IV.v = IV.v + IV.w$; $IV.w = IV.v - IV.w$; $IV.v = IV.v - IV.w$

Sorting actions for any node v

ABB	$S = B \wedge check(l_v, A) \wedge check(r_v, B)$	$\longrightarrow$	$swap(v, r_v)$; $S = A$
BAA	$S = A \wedge check(l_v, B) \wedge check(r_v, A)$	$\longrightarrow$	$swap(v, r_v)$; $S = B$

Figure 6: *Algorithm $\mathcal{A}\_\mathcal{S}_1$ (EREW Model)*

Algorithm $\mathcal{A}\_\mathcal{S}_1$ is deterministic.

5.2 Sorting in the Shared Memory Model using Constant Space

In Algorithm $\mathcal{S}_1$ (Figure 7), a node v holds three variables: variable IV to be sorted, a variable $S \in \{A, B, X, Y\}$, and a variable $tmpS \in \{A, B\}$. Variable $tmpS$ stores the value of variable S temporarily while the swap is performed.

For some node v, let $S = S.v$, $IV = IV.v$, $tmpS = tmpS.v$, $S_l = S.l_v$, $IV_l = IV.l_v$, $S_r = S.r_v$, $IV_r = IV.r_v$. Macro $swap'(v, r_v, value)$ executes the first step of swapping between node v and its right node r_v, and the value $value$ to be given to variable $S.v$ after the swap is performed is stored in variable $tmpS.v$. Predicate $check(v)$ has been defined in Section 3.

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).

In order to perform the swap, nodes v and r_v need to change their variables S, from either A or B to either X or Y. Since node v will change the value of its S after the swap, the future value of $S.v$ and the value of $S.r_v$ are stored in variables $tmpS.v$, respectively $tmpS.r_v$, by each node. Node v

Macro $swap'(v, w, tS)$ ::
if $(w \neq \perp \wedge IV.v > IV.w)$ then $tmpS.v = tS$; $IV.v = IV.v + IV.w$; $S.v = X$

Sorting actions for any node v

ABB	$S = B \wedge check(l_v, A) \wedge check(r_v, B)$	$\longrightarrow$	$swap'(v, r_v, A)$
BAA	$S = A \wedge check(l_v, B) \wedge check(r_v, A)$	$\longrightarrow$	$swap'(v, r_v, B)$

Synchronizing actions for any node v

$S1$	$S \in \{A, B\} \wedge l_v \neq \perp \wedge S_l = X$	$\longrightarrow$	$IV = IV_l - IV$; $tmpS = S$; $S = Y$
$S2$	$S = X \wedge r_v \neq \perp \wedge S_r = Y$	$\longrightarrow$	$IV = IV - IV_r$; $S = tmpS$
$S3$	$S = Y \wedge l_v \neq \perp \wedge S_l \neq X$	$\longrightarrow$	$S = tmpS$
$C1$	$S = Y \wedge l_v = \perp$	$\longrightarrow$	$S = tmpS$
$C2$	$S = X \wedge r_v = \perp$	$\longrightarrow$	$S = tmpS$
$C3$	$S = X \wedge r_v \neq \perp \wedge S_r = X$	$\longrightarrow$	$S = tmpS$

Figure 7: *Algorithm $\mathcal{S}_1$ (Shared Memory Model)*

changes its S to X (macro $swap'$) and node r_v changes its S to Y (Guard $S1$). The swap started by node v already in macro $swap'$ is continued by node r_v in Guard $S1$, and finished by node v in Guard $S2$ (where it also restores its S). Once the swap is done, their S values are restored to A or B, node v is in Guard $S2$, and node r_v is in Guard $S3$.

In Figure 8, nodes v and r_v need to swap their values. The *state* of a node is an ordered triple, $(S, IV, tmpS)$.

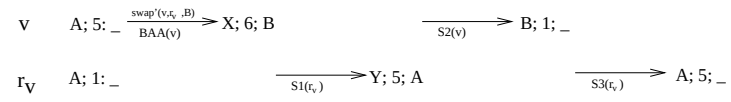


Figure 8: Nodes v and r_v swap their IV values

Algorithm $\mathcal{S}_1$ is deterministic also.

5.3 Sorting in the Shared Memory Model using an Extra Variable

In Algorithm $\mathcal{S}_2$ (Figure 9), a node v holds three variables: variable IV to be sorted, a variable $S \in \{A, B\}$, and a variable $tmpIV$. Swapping between values $IV.v$ and $IV.r_v$, when unsorted, is done in three steps as follows: $tmpIV.v \leftarrow IV.v$, $IV.v \leftarrow IV.r_v$, $IV.r_v \leftarrow tmpIV.v$.

For some node v, let $S = S.v$, $IV = IV.v$, $tmpIV = tmpIV.v$, $S_l = S.l_v$, $IV_l = IV.l_v$, $S_r = S.r_v$, $IV_r = IV.r_v$.

Macro macro $sort(v, l_v, r_v)$ contains two if-statements. The first statement (called L_1) checks whether node l_v has started the swap with node v (Steps 1. and 2. have been performed at node l_v); if yes, then node v executes Step 3. The second statement (called L_2) checks whether node v needs to swap with node r_v; if yes, Steps 1. and 2. are performed.

Macro $sort(v, u, w)$ ::
if $(u \neq \perp \wedge IV.u = IV.v \wedge IV.u < tmpIV.u)$ then $IV.v = tmpIV.u$ /* statement L1 */
if $(w \neq \perp \wedge IV.v > IV.w)$ then $tmpIV.v = IV.v$; $IV.v = IV.w$ /* statement L2 */

Sorting actions for any node v

ABB $S = B \wedge check(l_v, A) \wedge check(r_v, B) \longrightarrow sort(v, l_v, r_v);\ S = A$

BAA $S = A \wedge check(l_v, B) \wedge check(r_v, A) \longrightarrow sort(v, l_v, r_v);\ S = B$

Figure 9: *Algorithm $\mathcal{S}_2$ (Shared Memory Model)*

Assume that node v and its right neighbor r_v need to swap, and node v is enabled. Then node v executes the macro *sort*, stores in $tmpIV.v$ its value $IV.v$ and in $IV.v$ the value of $IV.r_v$, and becomes disabled. When node r_v becomes enabled, it will store in $IV.r_v$ the value of $tmpIV.v$ and then compares it with the value of its right neighbor, and so on.

In Figure 10, nodes v and r_v need to swap their values. The *state* of a node is an ordered triple, $(S, IV, tmpIV)$.

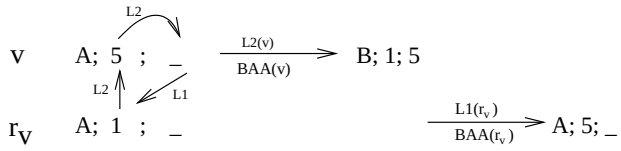


Figure 10: Nodes v and r_v swap their IV values

Algorithm $\mathcal{S}_2$ is deterministic also.

6 Algorithm $\mathcal{SSDS}$ as a Read/Write Atomicity Protocol

The authors of [12] distinguish between *composite* and *read/write* atomicity algorithms as follows. The guards of the guarded actions in a read/write atomicity algorithm include either only the node's variables or the node's variables of its neighbors but not both (mixed). A composite atomicity

algorithm contains at least one mixed guard; the vast majority of the self-stabilizing algorithms are composite.

Algorithm $\mathcal{SSDS}$ is composite and in this section we show why it does not work under the read/write atomicity model without adding other variables, thus increasing the memory requirements per node.

A node remembers three values: its own, and a copy of each of its neighbors'. The node's own value is represented as a capital letter, the copies of its neighbors' as small letters to the left and right. The end nodes remember only two variables. For example, if a node's own value is A, its copy of its left neighbor's value is B, and its copy of its right neighbor's value is A, we write the node as: bAa.

A global configuration is represented by a string over $\{A, B, a, b\}$. We define the following two codes:

- *Node codes.* Each node is represented by a string of two symbols if it is an end node, three symbols otherwise.

  The regular expression for the left node's code is (A+B)(a+b). The regular expression for the right node's code is (a+b)(A+B). The regular expression for any other node's code is (a+b)(A+B)(a+b).

  The global code string is the concatenation of the node codes. Here is an example global code string: AabAbbAaaBbaA In this example, the node codes are: Aa, bAb, bAa, aBb, aA.

- *Edge codes.* An edge code is the four-symbol substring of the code string starting and ending with either A or B. The regular expression for an edge code is (A+B)(a+b)(a+b)(A+B). In the example, the edge codes are: AabA, AbbA, AaaB, BbaA.

For each of the two codes, we define grammars as follows:

- *Node grammar.*

  We define the following *node grammar*, where symbol * refers to an arbitrary symbol that remains unchanged during the replacement step:

  $Aa \rightarrow Ba$

  $Bb \rightarrow Ab$

  $bA \rightarrow bB$

  $aB \rightarrow aA$

  $bAa \rightarrow bBa$

$aBb \rightarrow aAb$

$*a \rightarrow *b$

$*b \rightarrow *a$

$a* \rightarrow b*$

$b* \rightarrow a*$

$** a \rightarrow ** b$

$** b \rightarrow ** a$

$a** \rightarrow b**$

$b** \rightarrow a**$ There are actually 30 different replacement rules in the node grammar, since each * could represent either of two choices.

- *Edge grammar.*

  We define the following *edge grammar*, where symbol * refers to an arbitrary symbol that remains unchanged during the replacement step:
  $Aa** \rightarrow Ba**$

  $Bb** \rightarrow Ab**$

  $A*b* \rightarrow A*a*$

  $B*a* \rightarrow B*b*$

  $*a*B \rightarrow *b*B$

  $*b*A \rightarrow *a*A$

  $**aB \rightarrow **aA$

  $**bA \rightarrow **bB$

  There are actually 32 different replacement rules in the edge grammar, since each * could represent either of two choices.

A change in the global code is permitted in one step (do not confuse "step" with "round") if and only if every edge code substring either does not change or is replaced using a rule of the edge grammar, and every node code substring either not change or is replaced using a rule of the node grammar.

For example, AabAbbAaaBbaA may change to AaaAbbBaaBbbA, since all the following substring changes are permitted:
$AabA \rightarrow AaaA$
$AbbA \rightarrow AbbB$
$AaaB \rightarrow BaaB$
$BbaA \rightarrow BbbA$

$Aa \rightarrow Aa$
$bAb \rightarrow aAb$
$bAa \rightarrow bBa$
$aBb \rightarrow aBb$
$aA \rightarrow bA$

Here are changes that are allowed:

$***AabB** \rightarrow ***AbaB**$
$**bAa*Ab*A \rightarrow **bBa*Aa*A$
$**bAaaBb** \rightarrow **bBaaAb**$

Changes that you might think are allowed but are not:

$A*b*a*B \rightarrow A*a*b*B$
$***AaaBb** \rightarrow ***AbaAb**$

although each can be accomplished in two steps:

$A*b*a*B \rightarrow A*a*a*B \rightarrow A*a*b*B$
$***AaaBb** \rightarrow ***AbaBb** \rightarrow ***AbaAb**$

The edge codes can be divided into *good* and *bad* edges. Of the 16 possible edge codes, 8 are *good* and 8 *bad*:

AaaA good	AaaB bad	AabA bad	AabB bad	AbaA good	AbaB good
AbbA bad	AbbB good	BaaA good	BaaB bad	BabA good	BabB good
BbaA bad	BbaB bad	BbbA bad	BbbB good		

If an edge is bad, it can stay bad or become good. If an edge is good, it cannot become bad. If all edge codes of a global code are good, we say that the global code is good, otherwise we say the global code is bad.

In order for a read/write atomicity protocol based on the Algorithm $\mathcal{SSDS}$ to be self-stabilizing, we must show that: *(i) convergence* - Any bad global code will become good, and *(ii) closure* - A good global code cannot become bad.

We show that the convergence property does not hold in an asynchronous system. Specifically there is some initial global code string, such that, for any N, that the string does not become good after N rounds.

Consider the starting configuration AaaBbaBbbAabA. This configuration is bad (illegitimate), since all the nodes are enabled to enter CS (every edge is bad). Consider the following possible path of execution of some in the read/write atomicity protocol based on the Algorithm $\mathcal{SSDS}$ in an asynchronous system.

$AaaBbaBbbAabA \rightarrow BaaBbaAbbAabB \rightarrow BbaBbaAabAabB \rightarrow$
$BbaBbbAabAaaB \rightarrow BbaAbbAabBaaB \rightarrow BbaAabAabBbaB \rightarrow$

$BbbAabAaaBbaB \rightarrow AbbAabBaaBbaA \rightarrow AabAabBbaBbaA \rightarrow$
$AabAaaBbaBbbA \rightarrow AabBaaBbaAbbA \rightarrow AabBbaBbaAabA \rightarrow$
$AaaBbaBbbAabA$

Namely, after 12 steps, the execution ends in the starting configuration, without reaching a good (legitimate) state. Since every symbol in the string changes once in the first six steps, and once more in the next six steps, the sequence takes at least two rounds. We conclude that the convergence does not hold for this model.

7 Conclusion

In this paper, we present the first self-stabilizing distributed algorithm for a given synchronization problem on asynchronous oriented chains (Algorithm $\mathcal{SSDS}$). The algorithm is optimal in space complexity and asymptotically optimal in time complexity. We then give two applications of the proposed algorithm for oriented chains: a time and space optimal solution to the local mutual exclusion problem (Algorithm $\mathcal{LME}$), and distributed sorting. In solving distributed sorting, two shared memory self-stabilizing algorithms are proposed: a space and (asymptotically) time optimal solution (Algorithm $\mathcal{S}_1$), and an almost time optimal solution (Algorithm $\mathcal{S}_2$). All algorithms are self-stabilizing and uniform, and they work under the unfair distributed daemon.

Algorithm $\mathcal{SSDS}$ can be used to obtain optimal space solutions for other problems such as broadcasting, leader election, and mutual exclusion. It can also be extended to other oriented topologies such as oriented rings, rooted trees, directed acyclic graphs (DAG).

Acknowledgements

We thank the referees for valuable suggestions during the revision process. Based on their observations, the presentation of the paper has been greatly improved.

A short version of this paper has appear in the proceedings of the 2008 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, August 28-30, pages 303-306, 2008.

References

[1] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

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

[3] P Flocchini, E Kranakis, D Krizanc, F Luccio, and N Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel & Distributed Computing*, 64:254–265, 2004.

[4] A Sasaki. A time-optimal distributed sorting algorithm on a line network. *Information Processing Letters*, 83:21–26, 2002.

[5] A Sasaki. A time- and communication-optimal distributed sorting algorithm in a line network and its extension to the dynamic sorting problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, pages 444–453, 2004.

[6] C Boulinier, F Petit, and V Villain. When graph theory helps self-stabilization. *ACM Symposium on Principles of Distributed Computing*, pages 150–159, 2004.

[7] D Bein, AK Datta, and LL Larmore. Self-stabilizing space optimal synchronization algorithms on trees. In *Colloquium on Structural Information and Communication Complexity, LNCS 4056*, pages 334–348, 2006.

[8] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Workshop on Self-Stabilizing Systems*, pages 78–85, 1999.

[9] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *International Conference on Distributed Computing Systems*, pages 78–85, 2003.

[10] C Johnen, LO Alima, AK Datta, and S Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. *International Conference on Distributed Computing Systems (ICDCS)*, pages 487–494, 1999.

[11] A Arora and M Nesterenko. Stabilization-preserving atomicity refinement. *Journal of Parallel & Distributed Computing*, 62:766–791, 2002.

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