Self-stabilizing Synchronization Algorithms on Oriented Chains

Doina Bein*  Ajoy K. Datta†  Lawrence L. Larmore‡

Abstract

We present a space and (asymptotically) time optimal self-stabilizing scheme for a given synchronization problem on asynchronous oriented chains (Algorithm $SSDS$). The proposed scheme is uniform, and works under an unfair distributed daemon.

We use this scheme to solve two problems: local mutual exclusion and distributed sorting (where each process holds a single value and the values to be sorted are distinct). Algorithm $LM$ is a self-stabilizing, and space and time optimal solution to the local mutual exclusion problem — it uses two bits per node and stabilizes in $0$ rounds (it is snap-stabilizing). We propose two self-stabilizing solutions to distributed sorting: a space and asymptotically time optimal solution (Algorithm $S_1$), and an almost optimal time solution (Algorithm $S_2$). Algorithm $S_1$ uses three bits per node and stabilizes in $4(2n-2)$ rounds. Algorithm $S_2$ uses $L+1$ bits per node ($L$ is the maximum size of the initial values in the chain) and stabilizes in $2n-1$ rounds.

Algorithm $SSDS$ can be used to obtain optimal space solutions for other problems such as broadcasting, leader election, and mutual exclusion, and can be extended to other topologies.

Keywords: Distributed sorting, local mutual exclusion, oriented chain, self-stabilization, synchronization.

1 Introduction

Fault-tolerance is the ability of a system to withstand transient faults. A fault-tolerant system is guaranteed to still perform its function when a number of transient errors had occurred. In [1], Dijkstra defined a system as 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 distributed algorithms aim to achieve performance comparable to non-stabilizing distributed algorithms even if transient faults or arbitrary initializations cause the system to enter a state where a non-stabilizing algorithm may not continue to properly perform its task.

In this paper, we propose a general synchronization scheme for asynchronous oriented chains and use this scheme to solve two fundamental problems: sorting and local mutual exclusion.

Related Work. Distributed sorting (unrelated to node IDs) was studied in [2, 3, 4]. A time optimal ($n-1$ steps) solution to the distributed sorting problem in a synchronous oriented chain is given by Sasaki [3]. The space complexity in every node is $O(L)$, where $L$ is the maximal size of the initial values. Sasaki [4] also gives a time and communication optimal solution to the distributed sorting problem for an asynchronous oriented chain. The time complexity is $n-1$, and thus optimal, and the communication complexity is $n+2$. The algorithm uses three states and $O(L)$ space complexity per node. Both solutions lack the fault-tolerant feature, since they assume a correct initial state, and if a fault occurs, they run forever. Flocchini et al. [2] give an interesting analysis of the relationship between sorting and election in an anonymous asynchronous ring. A general scheme for solving any synchronization problem whose safety specification can be defined using a local property is given in [5].

Contributions. Our proposed contribution is threefold. We propose a general self-stabilizing scheme (Algorithm $SSDS$) for a given synchronization problem: “Given any $t > 0$, in finite time, every node is enabled

*School of Computer Science, University of Nevada, Las Vegas, NV 89154. Email: sioma@cs.unlv.edu.
†Contact author. School of Computer Science, University of Nevada, Las Vegas, NV 89154. Email: datta@cs.unlv.edu.
‡School of Computer Science, University of Nevada, Las Vegas, NV 89154. Email: larmore@cs.unlv.edu.
at least $t$ times, and whenever a node is enabled, all of its neighbors are disabled." Algorithm $SSDS$ is optimal in space complexity, i.e., it uses one bit space in every node. It 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 system, after $n - 1$ steps, every node is enabled every second round.)

We use the proposed scheme to solve the local mutual exclusion problem (Algorithm $LME$) 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. Algorithm $LME$ uses two bits per node and stabilizes in 0 rounds (it is snap-stabilizing [6, 7]).

For distributed sorting, we propose a space and time (asymptotically) optimal solution (Algorithm $S_1$), and an almost time optimal solution (Algorithm $S_2$). Algorithm $S_1$ sorts $n$ values in at most $4(2n - 2)$ rounds. It uses a total of 3 bits per node, and thus an improvement over the algorithms of [3, 4]. Algorithm $S_2$ sorts $n$ values in at most $2n - 1$ rounds (the minimum number of rounds required is $2n - 2$). It uses $L + 1$ bits per node, where $L$ is the maximum size of the initial values in the chain.

Outline of the paper. The model used in this paper and the synchronization problem are described in Section 2. Algorithm $SSDS$ and its proof of correctness are given in Section 3. Algorithm $LME$ is given in Section 4. Algorithms $S_1$ and $S_2$ are presented in Section 5. We first present a sorting algorithm for an abstract model of communication (Algorithm $A_{S_1}$), then show how sorting is done in the shared memory model of communication by using either a constant number of bits (Algorithm $S_1$) or $O(L)$ bits where $L$ is the maximal space required to store an initial variable (Algorithm $S_2$). We give concluding remarks in Section 6. In the appendix (Section A), we show why Algorithm $SSDS$ cannot be a read/write atomicity model without adding some other variables.

2 Preliminaries

We consider an asynchronous bi-directional, oriented chain of $n$ nodes. We use the terms of node and process interchangeably. Every node $v$ can distinguish between its left neighbor ($l_v$) and its right neighbor ($r_v$), and this left-right orientation is consistent among all the nodes in the network. If a node $v$ has only one neighbor, the value of the missing neighbor is represented as $\bot$. The leftmost node is denoted by $L$ and the rightmost node by $R$.

The model of communication between neighboring nodes is shared memory. A node can read and write its own memory, but can only read the memory of its neighbors. The program of every node consists of a finite set of guarded actions of the form: $<label>::=<guard>\rightarrow<action>$ that involve the node's variables and the variables of its neighbors. The state of a node is defined by the values of its variables. A system state, or (configuration), is simply a choice of a state for each node.

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. In a computation step, a distributed daemon selects a nonempty subset of enabled nodes. Each enabled node executes one of its enabled actions. The guard evaluation and the execution of the corresponding action are considered to be done in one atomic step.

We assume an asynchronous system. In order to compute the time complexity measure, we use the definition of a round [8]. A round is a minimal sequence of computation steps such that each processor that was enabled in the first configuration of the sequence has executed at least once during the sequence. We consider the unfair daemon: a continuously enabled process may not be selected for execution unless it is the only enabled process.

Given $C$, the set of all possible states, and a predicate $P$ over $C$, we denote by $L_P \subseteq C$ the set of all legitimate states with respect to $P$, or the set of all legitimate states when $P$ is understood.

Let $C_1$ and $C_2$ be subsets of $C$. $C_2$ is a closed attractor for $C_1$ if and only if the following conditions are true:
(i) for any initial state $c_1$ in $C_1$, for any execution $e$ in $E_{c_1}$ ($e = c_1, c_2, \ldots$), 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 $C_2$.

**Definition 1 (Self-Stabilization)** Given a predicate $P$, a system $S$ is called self-stabilizing with respect to $P$ (or simply, self-stabilizing, when $P$ is understood) if and only if $L_P$ is a closed attractor for $C$.

We consider the following synchronization problem in this paper:

**Definition 2 (Synchronization Problem)** Given any $t > 0$, find the minimum time $T$ such that every node is enabled at least $t$ times within $T$ time units, and whenever some node is enabled, no direct neighbor is enabled.

### 3 Algorithm $SSDS$

Algorithm $SSDS$ solves the synchronization problem defined in Definition 2. Each node holds a variable $S \in \{A, B\}$, thus needs only 1 bit. For any node $v$, let $S_v, S_l$, and $S_r$ represent the $S$-value of node $v$, $l_v$, and $r_v$, respectively. Predicate $check(v, s)$ checks if node $v$ exists, and if so, whether its $S$-value is $s$. Macro $execute(v)$ represents an application specific task that can only be executed if the node is enabled.

**Algorithm 3.1 Algorithm $SSDS$**

**Predicate** $check(v, s) \equiv (v = \bot \lor S_v = s)$

**Actions for any node $v$**

- $ABB$ \quad $S = B \land check(l_v, A) \land check(r_v, B) \rightarrow execute(v); S = A$
- $BAA$ \quad $S = A \land check(l_v, B) \land check(r_v, A) \rightarrow execute(v); S = B$

Actions $ABB$ and $BAA$ are enabled at node $v$ when the following two 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.

For example, given the starting configuration of a synchronous 7-node network shown in Figure 1(a)), the only enabled nodes are the odd-numbered nodes from left to right (first, third, fifth, and seventh). The next execution step brings the system into the configuration in Figure 1(b), in which the only enabled nodes are the even-numbered nodes. After one more step, the system reaches the configuration in Figure 1(c), followed by configuration in Figure 1(d) in one more step. The next configuration is the same as the initial configuration. This cycle repeats forever.

![Figure 1: Four steps in a synchronous system](image)

### 3.1 Proof of Correctness for $SSDS$

We show that Algorithm $SSDS$ stabilizes in at most $n - 2 + 2k$ rounds to the global predicate
\[k\text{-Exec} \equiv \{\forall k > 0 \exists T \text{ such that } \forall \text{ node } v, v \text{ has executed macro } \text{execute} \text{ at least } k \text{ times within } T \text{ rounds, and whenever } v \text{ is enabled, no neighbor is enabled}\}\]

and also works under the unfair distributed daemon (Property 7, Section 3.2).

We show that by executing Algorithm SDDS the following properties are true:
- Whenever a node is enabled, no neighbors are enabled (local mutual exclusion) (Property 1).
- At least one node is enabled (no deadlock) (Property 2).
- After a node executes, it remains disabled until all its neighbors execute (fairness) (Property 3).
- During the first \(n - 2 + 2k\) rounds, every node is enabled at least \(k\) times (no starvation) (Corollary 1).
Thus \(T = n - 2 + 2k\).

If \(n = 1\) (the distinguished node) and its starting state is \(A\), then it alternately executes Actions \(BAA\) and \(ABB\) forever. So, we will assume \(n > 1\) henceforth.

**Definition 3 (Configuration-String)** A configuration-string is defined as the sequence of the states (values of \(S\)) of all nodes from left to right.

Since there is a bijection between configurations and configuration-strings, by an abuse of notation, we simply say “configuration” to mean configuration-string.

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

**Definition 4** 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.

For example, the difference string of \(ABBAABBAAB\) or \(BAABBABBB\) is 101010100.

**Remark 1** Given a difference-string \(DS\) and a value \(S\) of some node, the corresponding configuration \(C\) is uniquely defined.

Given a node \(v\) and a configuration \(C\), let \(DS_C(v)\) be the substring of \(DS_C\) corresponding to its left neighbor (if any), itself, and its right neighbor (if any).

From the code of Algorithm SDDS, we observe that:

**Observation 1** Given any configuration \(C\):

(i) Guard ABB or BAA is enabled at the leftmost node \(L\), if and only if \(DS_C(L)\) is 0 and the execution of the guard changes it from 0 \(\to\) 1.

(ii) Guard ABB or BAA is enabled at the rightmost node \(v\), if and only if \(DS_C(v)\) is 1 and the execution of the guard changes it from 1 \(\to\) 0.

(iii) For \(n > 2\), Guard ABB or BAA is enabled at some node \(v\) other than leftmost or rightmost node, if and only if \(DS_C(v)\) is 10 and the execution of the guard changes it from 10 \(\to\) 01.

**Property 1** For any configuration \(C\) and for any node \(v\), if node \(v\) is enabled to execute, then neither node \(l_v\) nor \(r_v\) (if they exist) is enabled.

**Proof.** We have three cases.

1) \(v\) is the leftmost node \(L\). Then, from Observation 1(i), \(DS_C(v) = 0\), thus \(DS_C(r_v)\) starts with a 0. From Observation 1(ii, iii), for node \(r_v\) to be enabled to execute, \(DS_C(r_v) \in \{10, 1\}\), thus it should start with a 1. Contradiction.

2) \(v\) is the rightmost node \(R\). Then, from Observation 1(ii), \(DS_C(v) = 1\), thus \(DS_C(l_v)\) ends with a 1. From Observation 1(i, iii), for node \(l_v\) to be enabled to execute, \(DS_C(l_v) \in 0, 10\), thus it should end with a 0. Contradiction.
3) $v$ is a node other than $L$ or $R$. Then, from Observation 1(iii), $DSC(v) = 10$, thus $DSC(l_v)$ ends with a 1 and $DSC(r_v)$ starts with a 0. From Observation 1(i, iii), for node $l_v$ to be enabled to execute, $DSC(l_v) \in \{0, 10\}$, thus it should end with a 0. Contradiction. From Observation 1(ii, iii), for node $r_v$ to be enabled to execute, $DSC(r_v) \in \{10, 1\}$, thus it should start with a 1. Contradiction.

\[ \Box \]

**Property 2** In any configuration $C$ there exists at least one enabled node.

**Proof.** We have three cases:

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

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

3) If $DSC$ starts with a 1 and does not contain the substring 10, then it ends with a 1. By Observation 1(ii), the rightmost node $R$ is enabled to execute.

\[ \Box \]

**Property 3** For any node $v$, if node $v$ is enabled and it is selected to execute by the daemon, after the execution is completed, its actions are disabled.

**Proof.** From Observation 1(i, ii), if node $v$ is either the leftmost or the rightmost node, after executing its enabled guard, it becomes disabled. From Observation 1(iii), a node $v$ other than the leftmost or the rightmost node is enabled if $DSC(v)$ is 10. But once node $v$ executes and configuration $C$ changes to configuration $C'$, then $DSC'(v)$ is 01, so node $v$ is disabled.

Given two nodes $v$ and $r_v$ where $S.v = a$ and $S.r_v = b$, the notation “$a \leftarrow b$” denotes that state $b$ does not block state $a$ from being enabled (for $v$ to be enabled in state $a$, $S.r_v$ must be $b$). The notation $a \rightarrow b$ indicates that state $a$ does not block state $b$ from being enabled (for $r_v$ to be enabled in state $b$, $S.v$ needs to be $a$).

For example, the guards of Actions $BAA$ and $ABB$ can be re-written as $B \rightarrow A \leftarrow A$ and $A \rightarrow B \leftarrow B$, respectively. Note that a node is enabled only when its adjacent arrows point towards it.

We can use the above notation to define layers as follows. We start defining the layers of nodes from $L$ towards the right. Node $L$ is placed on some layer. 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. We can represent a configuration using this notation in a sawtooth-like level ordering. The peak nodes are the enabled nodes.

The difference-string of a given configuration is consistent with the orientations of the arrows between consecutive $S$ values (1 for $\nearrow$ and 0 for $\searrow$).

For example, for the configuration $BAAABBABBBBAABAA$ of a 16-node network, the diagram of the nodes is given in Figure 2(a).

(a) Some configuration in a layered arrangement

(b) Delay values for the nodes

Figure 2: Configuration $BAAABBABBBBAABAA$ of a 16-node network
Definition 5 (Node Delay) For each node \( v \) we define \( \text{delay}[v] \) to be an integer value between 0 and \( n-1 \) calculated recursively as follows: (i) there exists at least one node whose delay is 0, and (ii) if \( \text{delay}[u] = d \) and node \( v \) is a neighbor of node \( u \) such that \( S.v \rightarrow S.u \) then \( \text{delay}[v] = \text{delay}[u] + 1 \). If \( S.v \leftarrow S.u \) then \( \text{delay}[v] = \text{delay}[u] - 1 \).

For a chain of length \( n \), for any node \( v \), \( \text{delay}[v] \) is a value between 0 and \( n - 1 \), and defines also the layer of the node in the current configuration. The number of rounds that a node waits before it becomes enabled cannot exceed its delay value.

The delay values of the nodes in Figure 2(a) are given in Figure 2(b).

**Property 4** In any configuration, if \( w \) is a neighbor of \( v \) then \( \text{delay}[w] = \text{delay}[v] \pm 1 \).

**Proof.** From Definition 5.

An enabled node has all the adjacent arrows pointing towards it. After an enabled node executes, those arrows are reversed and the delay values need to be recalculated.

**Property 5** 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 \( v \) cannot execute its enabled guard for the \((t + 1)\)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 \((t + 1)\)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 Property 3, 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 \((t + 1)\)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 \( d_0 \) over all nodes. By the definition of array \( \text{delay} \), \( 0 \leq D_0 \leq n - 1 \).

**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 \( P(q) \) 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 \( q \geq 0 \) that Predicate \( P(q) \) holds.

1. **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.
2. **Inductive step** for \( q > 1 \), \( P(q - 1) \) 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 Property 4, \( d_0[u] = d_0[v] \pm 1 \). 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, \(P(q - 1)\) holds for every node, including node \(u\). Thus node \(u\) executes \(t\) times within \(q - 1\) rounds. From Property 5, 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, \(P(q - 1)\) holds for every node, including node \(u\). Thus node \(u\) executes \(t - 1\) times within \(q - 1\) rounds. From Property 5, 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. \(\square\)

**Corollary 1** For any value \(t > 0\), for any node \(v\) node \(v\) executes \(t\) times within the first \(n - 1 + 2t - 1\) rounds.

**Proof.** Follows from Lemma 1 since \(d_0[v] \leq n - 1\). \(\square\)

### 3.2 Unfair Distributed Daemon

In this section, we show that Algorithm 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 (Property 6). 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 7).

**Property 6** If a node \(v\) is enabled to execute but is not selected by the daemon, it remains enabled until it gets selected.

**Proof.** If some node \(v\) is enabled, by Property 1, neither of the existing neighbors is enabled. The neighboring nodes remain disabled until \(v\) executes. \(\square\)

**Property 7** 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 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.

Let \(A\) be the maximal set of nodes in the chain that execute infinitely often. Then \(v \notin A\). If node \(u\) executes infinitely often, then both neighbors of \(u\) execute infinitely often (Property 3, Lemma 1). Thus, if \(u \in A\), then \(left(u), right(u) \in A\). By induction, \(A\) consists of all nodes, contradiction. \(\square\)

### 4 Self-Stabilizing Local Mutual Exclusion Algorithm on Oriented Chains \(\mathcal{LME}\)

Each node holds two variables: 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.
Algorithm 4.1 Algorithm $\mathcal{LME}$

\[
ABB \quad S = B \land \text{check}(l_v, A) \land \text{check}(r_v, B) \quad \rightarrow \quad \text{if request then } CS; \text{ request } = false
\]

\[
S = A
\]

\[
BAA \quad S = A \land \text{check}(l_v, B) \land \text{check}(r_v, A) \quad \rightarrow \quad \text{if request then } CS; \text{ request } = false
\]

\[
S = B
\]

A protocol solves the local mutual exclusion problem if any configuration of the system running the protocol has two properties ([9]): (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.

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

5 Self-Stabilizing Distributed Sorting Algorithms on an Oriented Chain

In this section we present three algorithms for the distributed sorting problem in an oriented chain: $\mathcal{A}_S_1$ (in Section 5.1), $S_1$ (in Section 5.3), and $S_2$ (in Section 5.4). Algorithm $\mathcal{A}_S_1$ is implemented in an abstract model. Algorithms $S_1$ and $S_2$ are implemented in the shared memory model. The correctness proof for Algorithm $\mathcal{A}_S_1$ is given in Section 5.2. The proof of correctness for Algorithm $S_2$ is given in Appendix B.1, and the reduction of Algorithm $S_1$ to Algorithm $\mathcal{A}_S_1$ is given in Appendix B.2.

Let $x$ and $y$ be two values to be swapped. Besides the usual swap of two values using an extra variable, swapping can also be done in three steps without using an extra variable, as follows:

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

5.1 Distributed Sorting on an Oriented Chain

Each node, besides the variable $S$, holds one variable $IV$ to be sorted. Algorithm $\mathcal{A}_S_1$ (Figure 5.1) is a particular case of Algorithm $SSDS$, in which the macro $execute(v)$ is replaced by the macro $swap(v, r_v)$ that swaps the values $IV_v$ and $IV_r_v$.

Consider an abstract model, different from the shared memory model, in which a node $v$, in order to execute the swap, can modify the right neighbor variable $IV_r_v$.

Intuitively, since by executing Algorithm $SSDS$, local mutual exclusion is satisfied in any configuration (see Property 1), a node can synchronize the swap with its right neighbor. We assume for now that the swap is done in an atomic step (macro $swap$), and we show in Sections 5.3 and 5.4 how this is done in the shared memory model.

For a node $v$, let $S = S_v, S_i = S_{l_v}, S_r = S_{r_v}$. Predicate $check(v)$ has been defined in Section 3.

Algorithm 5.1 Algorithm $\mathcal{A}_S_1$

Macro $swap(v, w)$ :: if $(w \neq \perp \land IV_v > IV_w)$ then $IV_v = IV_v + IV_w$; $IV_w = IV_v - IV_w$; $IV_v = IV_v - IV_w$

Sorting actions for any node $v$

\[
ABB \quad S = B \land \text{check}(l_v, A) \land \text{check}(r_v, B) \quad \rightarrow \quad \text{swap}(v, r_v); \quad S = A
\]

\[
BAA \quad S = A \land \text{check}(l_v, B) \land \text{check}(r_v, A) \quad \rightarrow \quad \text{swap}(v, r_v); \quad S = B
\]
5.2 Proof of Correctness of $A_\mathcal{S}_1$

Assume that the position of node $L$ is 1, and the position of the rightmost node is $n$.

Besides local mutual exclusion, sorting 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, we define a pseudo-time function $\Psi$ from the set of nodes in the network to non-negative integers that describes when certain event (comparison) will be executed between the node and its right neighbor. This function is computed recursively from the previous configuration, starting with the initial configuration.

Let $\Psi_0$ be the function for the starting configuration $C_0$ defined as follows: (i) the leftmost node $L$ has the same $\Psi$ value as its right neighbor, i.e. $\Psi_0(1) = \Psi_0(2)$, and (ii) given two neighboring nodes $v$ and its left neighbor $l_v$ with positions $i$ and $i-1$, $\Psi_0(i) = \frac{d_0[v] + \phi_0[l_v] - 1}{2}$.

For example, given the configuration in Figure 2(a), the $\Psi_0$ values are given in Figure 3(a).

![Diagram](a) Initial configuration  
(b) After execution of the marked nodes

Figure 3: Pseudo-time values

If the node at position $i$ is enabled, then $\Psi_0(i) = \Psi_0(i + 1)$ (if $i \leq n - 1$).

**Definition 6** Let $\Psi_j$ and $\Psi_{j+1}$ be the pseudo-time functions for two consecutive configurations in some execution $C_j \Rightarrow C_{j+1}$. The function $\Psi_{j+1}$ is computed as follows:

- if the node at position $i$ has executed during this step then $\Psi_j(i)$ and $\Psi_j(i + 1)$ increase by 1:
  
  $\Psi_{j+1}(i) = \Psi_j(i) + 1$ and $\Psi_{j+1}(i + 1) = \Psi_j(i + 1) + 1$. Additionally, if node $r_L$ executes, then node $L$ must also increase its pseudo-time, i.e. $\Psi_{j+1}(1) = \Psi_j(1) + 1$.

- all other nodes keep their current pseudo-time values $\Psi_{j+1}(k) = \Psi_j(k)$.

For example, given $\Psi_0$ from Figure 3(a), if the marked nodes execute, then the next pseudo-time values are the ones in Figure 3(b).

**Observation 2** The following relations are true:

(i) $\Psi_0(1) \leq n - 2$

(ii) For any $2 \leq i \leq n$,

$$\Psi_0(i) \leq \max \left\{ \frac{n - i}{2}, \frac{n - i}{i - 2} \right\}$$

**Corollary 2** $\Psi_0(i) \leq n - 2$, for any $i$, $1 \leq i \leq n$.

Let $\mathcal{E}(i, t)$ be the predicate: node $v$ at position $i$ is enabled if $\Psi(i) = t$, $1 \leq i \leq n$.

Define $\text{PARITY}$ as follows:

$$\text{PARITY} = \begin{cases} 0, & \text{if } \mathcal{E}(i, t) \text{ and } i + t \text{ is even for some } i \text{ and some } t \\ 1, & \text{if } \mathcal{E}(i, t) \text{ and } i + t \text{ is odd for some } i \text{ and some } t \end{cases}$$
Observation 3 \textit{PARITY is a global constant.}

Property 8 \( \mathcal{E}(i,t) \) holds if and only if \( t \geq \Psi_0(i) \) and \( i + t + \text{PARITY} \) is even.

Proof. From Definition 6 we observe that:
- If \( \mathcal{E}(i,t) \) is true then \( \mathcal{E}(i,t+2k+1) \) is false and \( \mathcal{E}(i,t+2k) \) is true, for all \( k \geq 0 \).
- If \( \mathcal{E}(i,t) \) is false then \( \mathcal{E}(i,t+2k+1) \) is true and \( \mathcal{E}(i,t+2k) \) is false, for all \( k \geq 0 \).

\( \square \)

Property 9 Given a starting configuration \( C_0 \) and some configuration \( C_j \) after Algorithm SSDS has executed a number of steps, then the number of rounds \( q \leq \min\{1 \leq i \leq n, \Psi_j(i)\} \).

Proof. A round has elapsed if all the enabled nodes have increased their \( \Psi \) values by at least one unit, thus the minimum value among them has increased at least by one.

We use the definition of a \textit{rank} ([10]). The \textit{rank} of an element in a set is equal to its position in a non-descending order of the set. Since we assume that the values are not necessarily distinct, two equal value elements may have different ranks. Even so, we show that in linear time, the values in the sorted network arrange in increasing order of their rank; thus the oriented chain becomes sorted.

We define the array \( \text{pos} \) with two parameters as follows.

Definition 7 Given \( r, 1 \leq r \leq n, \) and \( t \geq 0 \), \( \text{pos}[r,t] = i \) if the node at position \( i \) holds the value of rank \( r \) at the time that the current \( \Psi(i) = t \).

If initially, the element of some rank \( r \) is at position \( i \) and \( \Psi_0(i) = t_0 \), then we assume that for any \( t \), \( 0 \leq t \leq t_0 \), \( \text{pos}[r,t] = \text{pos}[r,t_0] \).

Lemma 2 proves that within \( 2n-2 \) pseudo-steps, the values in the chain are sorted.

Lemma 2 For any chain with \( n \) nodes, \( n > 1 \), we have the following:
(a) At any pseudo-time \( t \geq n-2 \), alternated nodes are enabled:
(b) If alternated nodes are enabled at pseudo-time \( t = 0 \), then after at most \( n \) pseudo-steps the values are sorted: \( \text{pos}[r,n] = r \), for all ranks \( r \in 1 \ldots n \).

Proof. (a) For any position \( i \in 1 \ldots n \), for any \( t \geq \Psi_0(i) \), Predicate \( \mathcal{E}(i,t) \) is true if and only if \( i + t + \text{PARITY} \) is even.

Since \( \forall i \in 1..n, \Psi_0(i) \leq n-2 \), it results that \( \forall t \geq n-2 \), Predicate \( \mathcal{E}(i,t) \) is true if and only if \( i + t + \text{PARITY} \) is even. Thus if \( t + \text{PARITY} \) is even, then all even-position nodes are enabled. Otherwise all odd-position nodes are enabled.

(b) Let \( \mathcal{P}(n) \) be the predicate: "Any chain of length \( n \) where alternated nodes are enabled becomes sorted after at most \( n \) pseudo-steps." We show by induction on \( n \geq 2 \) that Predicate \( \mathcal{P}(n) \) holds.

Let \( S.t. \) and \( IV.t. \) be the value of variable \( S \), respectively \( IV \), of the node at position \( i \) in the \( n \)-node chain at pseudo-time \( t \).

Basic step \( n = 2 \). If the values are unsorted (\( IV.1.0 > IV.2.0 \)), then in at most two steps they become sorted (\( IV.1.2 > IV.2.2 \)).

Inductive step. Predicate \( \mathcal{P}(n-1) \) is true and we show that Predicate \( \mathcal{P}(n) \) is true. We assume that at time \( t = 0 \) all alternated nodes are enabled and let \( C \) be such a configuration.

Property 10 shows that the maximum value moves to the last position.

Property 10

For any \( t \geq 2 \), \( \text{pos}[n,t] = \min \left\{ \begin{array}{ll} \text{pos}[n,t-1] + 1, & \text{if pos}[n,t-1] < n \\ n, & \text{if pos}[n,t-1] = n \end{array} \right. \)
Proof. Let $p$ be the position of the maximum element at pseudo-time $0$: $pos[n,0] = p$. If $p = n$ then $\forall t \geq 0, pos[n,t] = n$. Assume $p < n$. Predicate $\mathcal{E}(p,0)$ can be either true or false.

- Predicate $\mathcal{E}(p,0)$ is true ($p + 0 + \text{parity}$ is even). Then the node at position $p$ executes, and the value of rank $n$ moves one position closer to the end of the chain: $pos[n,1] = pos[n,0] + 1 = p + 1$. Then $\forall t \geq 2$ such that $p + t \leq n$, Predicate $\mathcal{E}(p + t, t)$ is true, and $pos[n,t] = pos[n,t-1] + 1$, if $pos[n,t-1] < n$.

- Predicate $\mathcal{E}(p,0)$ is false ($p + 0 + \text{parity}$ is odd). Then Predicate $\mathcal{E}(p,1)$ is true; the node at position $p$ executes, and the value of rank $n$ moves one position closer to the end of the chain: $pos[n,2] = pos[n,1] + 1 = p + 1$. Then $\forall t \geq 2$ such that $p + t \leq n$, Predicate $\mathcal{E}(p + t - 1, t)$ is true, and $pos[n,t] = pos[n,t-1] + 1$.

Corollary 3 $pos[n,n] = n$.

We now define an instance of size $n-1$ of a chain. By our induction hypothesis that will be sorted in at most $n-1$ pseudo-steps. We will be able to conclude that the values in the $n$-node chain in configuration $C$ will be sorted in at most $n$ pseudo-steps.

Let $C'$ be the configuration of a $(n-1)$-node chain obtained from the $n$-node chain by removing the maximum value. We show that in configuration $C'$ alternative nodes are enabled.

Let $S'.i.t$ and $IV'.i.t$ be the value of variable $S$, respectively $IV$, of the node at position $i$ in the $(n-1)$-node chain at pseudo-time $t$. Configuration $C'$ is defined as follows:

\[
IV'.i.0 = \begin{cases} 
IV.i.1, & \text{if } i < pos[n,1] \\
IV.(i+1).0, & \text{if } i + 1 > pos[n,0]
\end{cases}
\]

\[
S'.i.0 = \begin{cases} 
S.i.1, & \text{if } i < pos[n,1] \\
\text{reverse}(S.(i+1).0), & \text{if } i + 1 > pos[n,0]
\end{cases}
\]

Predicate $\mathcal{E}'(i,t)$ is: ”The node at position $i$ in the $(n-1)$-node chain is enabled at the pseudo-time $t.”$ We show in Property 11 that Predicate $\mathcal{E}'(i,0)$ depends on $\mathcal{E}(i,1)$ and $\mathcal{E}(i+1,0)$.

Property 11

Predicate $\mathcal{E}'(i,0) \equiv \begin{cases} 
\mathcal{E}(i,1), & \text{if } i < pos[n,1] \\
\mathcal{E}(i+1,0), & \text{if } i + 1 > pos[n,0]
\end{cases}$

Proof. It follows from definition of $S'.i.0$. □

Corollary 4 In configuration $C'$ alternated nodes are enabled.

Let \text{parity}' be a value such that a node at position $i$ in the $(n-1)$-node chain is enabled at pseudo-time $t$ if $i + t + \text{parity}'$ is even. \text{parity}' is a global constant.

Property 12 \text{parity}' = 1 - \text{parity}.

Proof. For $i < pos[n,1]$, Predicate $\mathcal{E}'(i,0) \equiv \mathcal{E}(i,1)$. Thus Predicate $\mathcal{E}'(i,0)$ is true if and only if $i + 1 + \text{parity}$ is even.

For $i + 1 > pos[n,0]$, Predicate $\mathcal{E}'(i,0) \equiv \mathcal{E}(i+1,0)$. Thus Predicate $\mathcal{E}'(i,0)$ is true if and only if $i + 1 + \text{parity}$ is even.

Thus $\forall i \in 1\ldots n-1$, Predicate $\mathcal{E}'(i,0)$ is true if and only if $i + 1 + \text{parity}$ is even, or $i + 1 - \text{parity}$ is even. □
Property 13 For all $t \geq 0$,
\[
E'(i, t) = \begin{cases} 
E(i, t + 1), & \text{if } i < pos[n, t + 1] \\
E(i + 1, t), & \text{if } i + 1 > pos[n, t] 
\end{cases}
\]
\[
S'.i.t = \begin{cases} 
S.i.(t + 1), & \text{if } i < pos[n, t + 1] \\
reverse(S.(i + 1).t), & \text{if } i + 1 > pos[n, t] 
\end{cases}
\]
\[
IV'.i.t = \begin{cases} 
IV.i.(t + 1), & \text{if } i < pos[n, t + 1] \\
IV.(i + 1).t, & \text{if } i + 1 > pos[n, t] 
\end{cases}
\]

Proof. By induction on $t \geq 0$.

Basic step $t = 0$. It results from the definition of $IV'.i.0$, $S'.i.0$, and Property 11.

Inductive step $t > 0$.

Predicate $E'(i, t)$ is true if and only $i + t + \text{PARITY}'$ is even, or $i + t + 1 + \text{PARITY}'$ is odd, equivalent to $i + t + 1 + \text{sc parity}$ is even. Thus Predicate $E'(i, t)$ is equivalent to Predicate $E(i, t + 1)$ when $i < pos[n, t + 1]$, and also equivalent to Predicate $E(i, t + 1)$ when $i + 1 > pos[n, t]$.

Without loss of generality assume that at pseudo-time $t - 1$ the odd-position nodes are enabled in the $n$-node chain. By induction hypothesis, it results that the even-position nodes in the $(n - 1)$-node chain are enabled at pseudo-time $t - 1$. Thus:

\[
S'.i.t = \begin{cases} 
S'.i.(t-1), & \text{if } i \text{ is even} \\
reverse(S'.i.(t-1)), & \text{if } i \text{ is odd} 
\end{cases}
\]

and

\[
IV'.i.t = \begin{cases} 
IV'.i.(t-1), & \text{if } i \text{ is even and } IV'.i.(i + 1).(t - 1) \geq IV'.i.(t - 1) \\
IV.(i+1).(t-1), & \text{if } i \text{ is even and } IV'.i.(i + 1).(t - 1) < IV'.i.(t - 1) \\
IV'.i.(t-1), & \text{if } i \text{ is odd and } IV'.i.(t - 1) \geq IV'.i.(i - 1).(t - 1) \\
IV.(i-1).(t-1), & \text{if } i \text{ is odd and } IV'.i.(t - 1) < IV'.i.(i - 1).(t - 1) 
\end{cases}
\]

At pseudo-time $t - 1$, all odd-position nodes in the $n$-node chain are enabled, thus:

\[
S.i.t = \begin{cases} 
S.i.(t-1), & \text{if } i \text{ is even} \\
reverse(S.i.(t-1)), & \text{if } i \text{ is odd} 
\end{cases}
\]

and

\[
IV.i.t = \begin{cases} 
IV.i.(t-1), & \text{if } i \text{ is odd and } IV.(i + 1).(t - 1) \geq IV.i.(t - 1) \\
IV.(i+1).(t-1), & \text{if } i \text{ is odd and } IV.(i + 1).(t - 1) < IV.i.(t - 1) \\
IV.i.(t-1), & \text{if } i \text{ is even and } IV.i.(t - 1) \geq IV.(i - 1).(t - 1) \\
IV.(i-1).(t-1), & \text{if } i \text{ is even and } IV.i.(t - 1) < IV.(i - 1).(t - 1) 
\end{cases}
\]

At pseudo-time $t$, all even-position nodes in the $n$-node chain are enabled, thus:

\[
S.i.(t + 1) = \begin{cases} 
S.i.t, & \text{if } i \text{ is odd} \\
reverse(S.i.t), & \text{if } i \text{ is even} 
\end{cases}
\]

and

\[
IV.i.t + 1 = \begin{cases} 
IV.i.t, & \text{if } i \text{ is even and } IV.(i + 1).t \geq IV.i.t \\
IV.(i+1).t, & \text{if } i \text{ is even and } IV.(i + 1).t < IV.i.t \\
IV.i.t, & \text{if } i \text{ is odd and } IV.i.t \geq IV.(i - 1).t \\
IV.(i-1).t, & \text{if } i \text{ is odd and } IV.i.t < IV.(i - 1).t 
\end{cases}
\]

We then calculate the value of $S'.i.t$ and $IV'.i.t$ when $i < pos[n, t]$ and it is either even or odd, when $i = pos[n, t]$ and it is either even or odd, and when $i > pos[n, t]$ and it is either even or odd.

By inductive step, at pseudo-time $t = n - 1$ the values in the $(n - 1)$-node chain are sorted: $IV'.i.t \leq IV'.i + 1.t, \forall i \in 1 \ldots n - 1$. Since $pos[n, n - 1] \geq n - 1$, by Property 13, $IV'.i.n - 1 = IV.i.n$, where $t = n - 1$. Thus the values in the $n$-node chain are sorted. □
5.3 Sorting in the Shared Memory Model using Constant Space

In Algorithm $S_1$ (Figure 5.3), 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 $S = S_v, IV = IV_v, tmpS = tmpS_v, S_i = S_i_v, IV_i = IV_i_v, S_r = S_r_v, IV_r = IV_r_v$. 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.

Algorithm 5.2 Algorithm $S_1$

Macro $swap'(v, w, tS)$ :: if $(w \neq \bot \land IV_v > IV_w)$ then $tmpS_v = tS$ ; $IV_v = IV_v + IV_w$ ; $S_v = X$

Sorting actions for any node $v$

$ABB \quad S = B \land check(l_v, A) \land check(r_v, B) \rightarrow swap'(v, r_v, A)$

$BAA \quad S = A \land check(l_v, B) \land check(r_v, A) \rightarrow swap'(v, r_v, B)$

Synchronizing actions for any node $v$

$S1 \quad S \in \{A, B\} \land l_v \neq \bot \land S_i = X \rightarrow IV = IV_i - IV ; tmpS = S ; S = Y$

$S2 \quad S = X \land r_v \neq \bot \land S_i = Y \rightarrow IV = IV - IV_r ; S = tmpS$

$S3 \quad S = Y \land l_v = \bot \rightarrow S = tmpS$

$C1 \quad S = Y \land r_v = \bot \rightarrow S = tmpS$

$C2 \quad S = X \land r_v \neq \bot \rightarrow S = tmpS$

$C3 \quad S = X \land r_v = \bot \land S_i = X \rightarrow S = tmpS$

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 Figure 4, nodes $v$ and $r_v$ need to swap their values. The state of a node is an ordered triple, $(S, IV, tmpS)$.

\[ v \quad A; 5;_\rightarrow \quad \frac{swap'(r_v, B)}{BAA(v)} \rightarrow X; 6; B \quad \frac{S2(v)}{S2(v)} \rightarrow B; 1; \rightarrow \]

\[ r_v \quad A; 1; \rightarrow \quad \frac{S1(r_v)}{Y; 5; A} \quad \frac{S3(r_v)}{A; 5; \rightarrow} \]

Figure 4: Nodes $v$ and $r_v$ swap their $IV$ values

5.4 Sorting in the Shared Memory Model using an Extra Variable

In Algorithm $S_2$ (Figure 5.4), 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:

1. $tmpIV_v \leftarrow IV_v$
2. $IV_v \leftarrow IV_r_v$
3. $IV_r_v \leftarrow tmpIV_v$

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.

13
Algorithm 5.3 Algorithm $S_2$

Macro $\text{sort}(v, u, w)$ : if $(u \neq \bot \land IV.u = IV.v \land IV.u < tmpIV.u)$ then $IV.v = tmpIV.u$ /* statement L1 */
if $(w \neq \bot \land IV.u > IV.w)$ then $tmpIV.v = IV.v$ ; $IV.v = IV.w$ /* statement L2 */

Sorting actions for any node $v$

$ABB$ $S = B \land \text{check}(l_v, A) \land \text{check}(r_v, B) \rightarrow \text{sort}(v, l_v, r_v); S = A$
$BAA$ $S = A \land \text{check}(l_v, B) \land \text{check}(r_v, A) \rightarrow \text{sort}(v, l_v, r_v); S = B$

Assume that node $v$ and its right neighbor $r_v$ need to swap, and node $v$ is enabled. Then node $v$ executes the macro $\text{sort}$, 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 5, nodes $v$ and $r_v$ need to swap their values. The state of a node is an ordered triple, $(S, IV, tmpIV)$.

![Figure 5: Nodes v and r_v swap their IV values](image_url)

6 Conclusion

In this paper, we present the first self-stabilizing distributed algorithm for a given synchronization problem on asynchronous oriented chains (Algorithm $SSDS$). 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 $LME$), and distributed sorting. In solving distributed sorting, two shared memory self-stabilizing algorithms are proposed: a space and (asymptotically) time optimal solution (Algorithm $S_1$), and an almost time optimal solution (Algorithm $S_2$). All algorithms are self-stabilizing and uniform, and they work under the unfair distributed daemon.

In proving the time complexity of the sorting, we introduce the notion of pseudo-time. Pseudo-time is similar to logical time introduced by Lamport [11].

References


A Appendix: Algorithm SSDS as a Read/Write Atomicity Protocol

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: bAb.

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:

  \[
  A \rightarrow A \mid A * A \\
  B \rightarrow B \mid B * B \\
  a \rightarrow a \mid a * a \\
  b \rightarrow b \mid b * b
  \]

  We define the following node grammar, where symbol * refers to an arbitrary symbol that remains unchanged during the replacement step:
\[ \begin{align*}
Aa & \rightarrow Ba \\
Bb & \rightarrow Ab \\
bA & \rightarrow bB \\
aB & \rightarrow aA \\
bAa & \rightarrow bBa \\
aBb & \rightarrow aAb \\
\ast a & \rightarrow \ast b \\
\ast b & \rightarrow \ast a \\
a\ast & \rightarrow b\ast \\
b\ast & \rightarrow a\ast \\
**a & \rightarrow **b \\
**b & \rightarrow **a \\
a** & \rightarrow b** \\
b** & \rightarrow a**
\end{align*} \]

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:

\[ \begin{align*}
Aa** & \rightarrow Ba** \\
Bb** & \rightarrow Ab** \\
A\ast b & \rightarrow A\ast a \\
B\ast a & \rightarrow B\ast b \\
\ast aB & \rightarrow \ast bB \\
\ast bA & \rightarrow \ast aA \\
**aB & \rightarrow **aA \\
**bA & \rightarrow **bB
\end{align*} \]

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, AabAbbAaaBbbaA may change to AaaAbbBaaBbbA, since all the following substring changes are permitted:

\[ \begin{align*}
AabA & \rightarrow AaaA \\
AbbA & \rightarrow AbbB \\
AaaB & \rightarrow BaaB \\
BbaA & \rightarrow BbbA \\
Aa & \rightarrow Aa \\
bAb & \rightarrow aAb \\
bAa & \rightarrow bBa \\
abB & \rightarrow aBb \\
aA & \rightarrow bA
\end{align*} \]
Here are changes that are allowed:

***AabB*** → ***AbaB***
**bAa*Ab*A → **bBa*Aa*A
**bAaaBb** → **bBaaAb**

Changes that you might think are allowed but are not:

A*b*a*B → A*a*b*B
***AaaBb** → ***AbaAb**

although each can be accomplished in two steps:

A*b*a*B → A*a*b*B
***AaaBb** → ***AbaBb** → ***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 AaaB 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 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 converge 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 SSDS in an asynchronous system. Namely, after 12 steps, the code returns to the original string. 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.

AaaBbaBbbAabA →
BaaBbaAbBbaB →
BbaBbaAaBabB →
BbaBbaAabAab →
BbaAbbAabBaa →
BbaAabAabBba →
BbbAabAaaBbb →
AabAabBaaBba →
AbaAabBbaBba →
AbaBbaAbBbb →
AbaBbaBbaAab →
AabBbaBbaAabA →
AaaBbaBbbAabA

The execution ends in the starting configuration, without reaching a good (legitimate) state. We conclude that the convergence does not hold for this model.
B Proofs of Correctness for Sorting Algorithms

B.1 Proof of Correctness of Algorithm $S_2$

By executing Algorithm $S_2$, a swap takes at most $1 + \epsilon$ rounds, where $\epsilon$ is a very small value. After $2n - 2$ rounds, the values are sorted, except maybe of a node that still has to take its $IV$-value from the $tmp.IV$-value of its left neighbor. Since that execution takes at most one additional round, the total time complexity for Algorithm $S_2$ is $2n - 1$ rounds, which is almost optimal.

B.2 Reduction of an Algorithm in Abstract Model to an Algorithm in Shared Memory Model

In this section we first show that Algorithm $S_1$ reduces to Algorithm $A_S$. We can then conclude that, starting from an arbitrary configuration, in at most $4(2n - 2)$ rounds, Algorithm $S_1$ sorts the values in non-decreasing order (Lemma 5).

**Definition 8 (Reduction)** Given two different models of communication $M_1$ and $M_2$, an algorithm $A_1$ in the model $M_1$ can be reduced to another algorithm $A_2$ in the model $M_2$ if there exists a one-to-many relation $R$ from the set of system configurations in the model $M_1$ to the set of the system configurations in the model $M_2$ such that the following conditions are true:

i) For each configuration of Algorithm $A_1$ in the model $M_1$ there exists at least one configuration of Algorithm $A_2$ in the model $M_2$.

ii) Lifting property Given $C_1$ and $C_2$ two configurations of Algorithm $A_1$ in the model $M_1$ such that $C_1 \rightarrow C_2$ is an execution step of Algorithm $A_1$, for any configuration $C_1' \in R(C_1)$, if Algorithm $A_2$ in the model $M_2$ starts in $C_1'$ there exists at least one execution path that starts in $C_1'$ and ends in some configuration $C_2' \in R(C_2)$.

If $A_1$ accomplishes a task in the model $M_1$ and $A_1$ reduces to $A_2$, then by Definition 8, $A_2$ accomplishes the same task in the model $M_2$.

We now show that Algorithm $S_1$ reduces to Algorithm $A_S$.

Let $S_v = (s_v, x_v)$ be the set of all variables of node $v$ in order $(state, IV)$ used by Algorithm $A_S$ in the abstract model. Let $S_v^w = (s_v, x_v, t_v)$ be the set of all variables of node $v$ in order $(state, IV, tmpState)$ used by Algorithm $S_1$ in the shared memory model.

Then $R$ is defined as follows. $R(S_1, ....S_n) = \{(s_1^{i_1}, ....S_n^{i_n}), t_i \in \{A, B\}, 1 \leq i \leq n\}$.

For each configuration $C_1$ of Algorithm $A_S$ in the abstract model, there exist $2^n$ configurations in $R(C_1)$ of Algorithm $S_1$ in the shared memory model; thus condition (i) of Definition 8 is satisfied.

We are left to show that condition (ii) of Definition 8 is satisfied (Lemma 3).

**Lemma 3** Given $C_1$ and $C_2$, two configurations of Algorithm $A_S$ in the abstract model, such that $C_1 \rightarrow C_2$ is an execution step of Algorithm $A_S$, for any configuration $C_1' \in R(C_1)$, if Algorithm $S_1$ in the shared memory model starts in $C_1'$, there exists at least one execution path that starts in $C_1'$ and ends in some configuration $C_2' \in R(C_2)$.

**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 consecutive nodes such that the first node in each chunk is enabled and there are no other enabled nodes in a chunk. If the first node or nodes are currently disabled, then the prefix of disabled nodes is not part of a chunk.

Given a configuration, there is a unique way to break it into chunks.
We need to show that an execution step of Algorithm $A_{\mathcal{S}_1}$ in the abstract model in one chunk affects only the nodes’ states in that chunk.

We know that if a node is enabled, then its neighbors (if any) are disabled. So, with the exception of the situation where the rightmost node is enabled, 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 in the next chunk. If the rightmost node is enabled, the last chunk contains only that node.

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

Let $C_1 = (H^1_1, H^2_2, \ldots, H^k_k)$ be the set of chunks, omitting the prefix of disabled nodes. Let $v$ the first node in some chunk $H^1_i$ of Configuration $C_1$ of Algorithm $A_{\mathcal{S}_1}$ in the abstract model. Assume without loss of generality that Action $BAA$ is enabled at $v$.

- If $i = k$, $H^1_k$ is the last chunk and contains only the rightmost node of the oriented chain, then even if the node $v$ is enabled, $v$ will just change state (from $A$ to $B$) without the swap (since it has no right neighbor to swap with).

The execution step of Algorithm $A_{\mathcal{S}_1}$ is: $((A, x_v)) \xrightarrow{\text{ BAA}(v)} ((B, x_v))$.

In the shared memory model this corresponds to: $((A, x_{v,v}), (A, y_v)) \xrightarrow{\text{ BAA}(v)} ((B, x_{v,v}), (A, y_v))$ that starts in any configuration of $\mathcal{R}(C_1)$ restricted to the chunk $H^1_i$, and ends in some configuration of $\mathcal{R}(C_2)$ restricted to the chunk $H^1_i$.

- If the chunk contains more than one node: $H^1_i = (S_v, S_{\text{right},v}, S_2, \ldots, S_{n_v})$.

For the ease of notation, assume $S_v = (A, x)$ and $S_{\text{right},v} = (A, y)$.

We have two cases, depending on whether $v$ and $\text{right},v$ have an inversion.

1) nodes $v$ and $\text{right},v$ do not have an inversion. Then the execution step of Algorithm $A_{\mathcal{S}_1}$ is:

$$((A, x), (s_1, y)) \xrightarrow{\text{ BAA}(v)} ((B, x), (s_1, y)).$$

In the shared memory model this corresponds to: $((A, x,), (A, y, )) \xrightarrow{\text{ BAA}(v)} ((B, x,), (A, y, ))$ that starts in any configuration of $\mathcal{R}(C_1)$ restricted to the chunk $H^1_i$, and ends in some configuration of $\mathcal{R}(C_2)$ restricted to the chunk $H^1_i$.

2) nodes $v$ and $\text{right},v$ have an inversion. Then the execution step of Algorithm $A_{\mathcal{S}_1}$ is: $((A, x), (A, y), S_2, \ldots, S_{n_v}) \xrightarrow{\text{ BAA}(v)} ((B, y), (A, x), S_2, \ldots, S_{n_v})$. ($v$ can affect only the variables of its right neighbor).

In the shared memory model this corresponds to:

$$((A, x,), (A, y, ), S_2^-, \ldots, S_{n_v}^-) \xrightarrow{\text{ BAA}(v)} ((X, x + y, B), (A, y, ), S_2^-, \ldots, S_{n_v}^-) \xrightarrow{S_1(\text{right},v)} ((B, y, ), (Y, x, A), S_2^-, \ldots, S_{n_v}^-) \xrightarrow{S_2(\text{right},v)} ((X, x + y, B), (Y, x, A), S_2^-, \ldots, S_{n_v}^-) \xrightarrow{S_3(\text{right},v)} ((B, y, ), (A, x, ), S_2^-, \ldots, S_{n_v}^-))$$

that starts in any configuration of $\mathcal{R}(C_1)$ restricted to the chunk $H^1_i$, and ends in some configuration of $\mathcal{R}(C_2)$ restricted to the chunk $H^1_i$.

If the start state of the node is either $A$ or $B$, then the value to be sorted is its initial value. If some node’s start state is either $X$ or $Y$, then it is possible for some of the three steps of the swap to be applied (see Section 5.1) 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.
Property 14 shows that for any node $v$ whose state is $X$, either the state remains $X$ and then $right.v$ will be in state $Y$ in at most three rounds (by executing Action $S1$), or $v$ changes its state to $A$ or $B$ in at most one round.

Property 15 shows that for any node $v$ such that $S.v = X \land S.(right.v) = Y$ then $IV.v$ receives the value of old value of $IV.(right.v)$ and then $v$ changes its state to $A$ or $B$, in at most one round. The node $right.v$ had already stored in $IV.(right.v)$ the old value of $IV.v$ (by executing Action $S1$) and by Property 16 will restore its state from $Y$ to either $A$ or $B$ (depending on the value of $tmpS$) in at most one round. We can then conclude that if the node state is either $X$ or $Y$, in at most four rounds it is in state $A$ or $B$ (Lemma 4).

**Property 14** For some node $v$ with $S.v = X$, either the state remains $X$ and then the $right.v$ node will be in state $Y$ in at most three rounds (by executing Action $S1$), or $v$ changes its state to $A$ or $B$ in at most one round.

**Property 15** For some node $v$ with $S.v = X$, Action $S2$ is enabled at $v$ if and only if $v$ has a right neighbor whose state is $Y$. Once Action $S2$ is executed at node $v$, $S.v$ becomes either $A$ or $B$.

**Property 16** If $S.v = Y$, then in at most two rounds, $S.v$ becomes either $A$ or $B$.

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

**Proof.** Directly from Properties 14, 15, and 16. \(\square\)

**Lemma 5** Starting from an arbitrary configuration, in at most $8n$ rounds, Algorithm $S_1$ arranges the $n$ values in non-decreasing order from left to right starting from the node $L$ and going right.

**Proof.** From Lemma 4, each swap takes at most 4 rounds. From Lemma 2, if a swap takes at most 1 round, then sorting takes at most $2n - 2$ rounds. Since the swap takes at most 4 rounds, we obtain a total of at most $4(2n - 2)$ rounds. \(\square\)