Distributed stabilizing data structures

Doina Bein
*University of Nevada, Las Vegas*

Follow this and additional works at: [https://digitalscholarship.unlv.edu/rtds](https://digitalscholarship.unlv.edu/rtds)

Repository Citation
[https://digitalscholarship.unlv.edu/rtds/2647](https://digitalscholarship.unlv.edu/rtds/2647)

This Dissertation is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitialscholarship@unlv.edu.
DISTRIBUTED STABILIZING DATA STRUCTURES

by

Doina Bein

Bachelor of Science
Al. I. Cuza University of Iasi, Romania
1996

Master of Science
Al. I. Cuza University of Iasi, Romania
1997

Master of Science
University of Nevada, Las Vegas
2001

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2006
The Dissertation prepared by

DOINA BEIN

Entitled

DISTRIBUTED STABILIZING DATA STRUCTURES

is approved in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

Ajoy Kumar Dutta
Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Examination Committee Member

Examination Committee Member
ABSTRACT

Distributed Stabilizing Data Structures

by

Doina Bein

Dr. Ajoy Kumar Datta, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Distributed algorithms aim to achieve better performance than sequential algorithms in terms of time complexity (or asymptotic time complexity) while keeping or lowering the memory requirement (space complexity) in a node. (In sequential algorithms, the memory requirement is the memory requirement of the algorithm itself.)

Self-stabilizing distributed algorithms aim to achieve a comparable performance to non-stabilizing distributed algorithms when transient faults or arbitrary initialization cause the system to enter a state where a non-stabilizing algorithm cannot continue to properly perform its task.

Transient faults can affect an existing data structure and alter its data content. As a result, the data structure may lose its properties, and the operations defined over the data structure will have unpredictable and undesirable results, making the data structure unusable.

We present several self or snap-stabilizing algorithms for particular data structures.

We propose an optimal self-stabilizing distributed algorithm for simultaneously activating non-adjacent processes on an oriented chain (Algorithm SSDS). We use Algorithm SSDS to accomplish two tasks: local mutual exclusion and line sorting. We propose two uniform, self-stabilizing, deterministic protocols on oriented chains: a time and space optimal solution to the local mutual exclusion problem (Algorithm LMEC), and a space and (asymptotic) time optimal solution to the distributed sorting problem (Algorithm...
We extend Algorithm SSSS to an asynchronous oriented ring with a distinguished node with some minor modifications, and we obtain general self-stabilization for simultaneously activated non-adjacent processes in an oriented ring with a distinguished process (Algorithm SSSSR). We use Algorithm SSSSR to accomplish two tasks: local resource allocation and ring sorting. We propose two uniform, self-stabilizing, deterministic protocols on oriented rings: a time and space optimal solution to the local resource allocation problem (Algorithm LRAAR), and a space and (asymptotic) time optimal solution to the distributed sorting problem (Algorithm SORT_r).

We extend Algorithm SSSS to an asynchronous rooted tree, and we obtain general self-stabilization for simultaneously activated non-adjacent processes in a rooted tree (Algorithm SSDST). We then give two applications of Algorithm SSDST: a time and space optimal solution to the local mutual exclusion problem (Algorithm LMET) and a space and (asymptotically) time optimal solution to the min heap problem (Algorithm HEAP).

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

We present the first snap-stabilizing distributed binary search tree (BST) algorithm. The proposed algorithm uses a heap algorithm (Algorithm Heap) as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem.
# TABLE OF CONTENTS

**ABSTRACT** ................................................................. iii  
**LIST OF FIGURES** ........................................................ vii  
**ACKNOWLEDGMENTS** ..................................................... viii  

**CHAPTER 1 INTRODUCTION** ........................................... 1  
   - Distributed Systems ...................................................... 1  
   - Topology and Communication Models .................................. 2  
   - Topology Models .......................................................... 2  
   - Communication Models ................................................... 4  
   - Distributed Protocols and Types of Daemons ....................... 4  
   - Self-Stabilization and Snap-Stabilization ........................... 6  
   - Related Work ................................................................... 7  
   - Self-Stabilizing Distributed Sorting .................................. 7  
   - Self-Stabilizing Heap and Binary-Search-Tree ..................... 10  
   - Self-Stabilizing Local Mutual Exclusion ............................ 11  
   - Orientation of a Ring .................................................... 12  
   - Contributions .............................................................. 12  
   - Organization of Dissertation .......................................... 16  

**CHAPTER 2 PRELIMINARIES** .......................................... 17  
   - Reducing an Algorithm in Abstract Model to an Algorithm in Shared-Memory Model ........................................... 17  
   - Algorithms for Orienting an Unoriented Chain ........................ 18  
   - Proof of Correctness for Algorithms $SENS_{1}^{\text{chain}}$ and $SENS_{2}^{\text{chain}}$ .................................................. 23  
   - Algorithms for Orienting an Unoriented Ring .......................... 26  

**CHAPTER 3 SIMULTANEOUSLY ACTIVATED PROCESSES ON A CHAIN** .................................................. 30  
   - Self-Stabilizing Distributed Simultaneous Execution of Non-adjacent Nodes on an Oriented Chain $SSDS$ .............................................. 30  
   - Proof of Correctness for Algorithm $SSDS$ ........................... 32  
   - Unfair Distributed Daemon ............................................. 37  
   - Self-Stabilizing Local Mutual Exclusion Algorithm on Oriented Chains $LMEC$ .................................................. 38  
   - Proof of Correctness of Algorithm $LMEC$ ............................ 39  
   - Algorithm $SSDS$ as a Read/Write Atomicity Protocol ............. 39  

**CHAPTER 4 OPTIMAL SORTING ON A CHAIN** ...................... 44  
   - Self-Stabilizing Distributed Sorting Algorithms in an Oriented Chain .......................................................... 44  
   - Distributed Sorting in an Oriented Chain ............................. 45  
   - Sorting in the Shared-Memory Model ................................ 45  
   - Proof of Correctness of Algorithm $ASORT_c$ ......................... 47  
   - Reduction of Algorithm $ASORT_c$ in Abstract Model to Algorithm $SORT_c$ in Shared-Memory Model ........................ 53  

**CHAPTER 5 SIMULTANEOUSLY ACTIVATED PROCESSES ON A RING** .................................................. 59  
   - Self-Stabilizing Distributed Simultaneously Execution of Non-adjacent Nodes on an Oriented Ring $SSDSR$ .............................................. 59  
   - Proof of Correctness for Algorithm $SSDSR$ .......................... 62  
   - Self-Stabilizing Local Resource Allocation Algorithm on Oriented Rings $LRAR$ .................................................. 72  

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 6  SORTING ON A RING ................................................................. 75
   Self-Stabilizing Distributed Sorting Algorithms in an Oriented Ring .......... 75
   Distributed Sorting in an Oriented Ring .................................................. 75
   Sorting in the Shared Memory Model ...................................................... 76
   Proof of Correctness of Algorithm $ASORT_r$ ........................................... 77
   Reduction of Algorithm $ASORT_r$ in Abstract Model to Algorithm $SORT_r$
   in Shared-Memory Model ....................................................................... 81

CHAPTER 7  SIMULTANEOUSLY ACTIVATED PROCESSES ON A TREE .... 83
   Self-Stabilizing Distributed Simultaneous Execution of Non-adjacent Nodes
   in a Rooted Tree $SSDST$ ........................................................................ 83
   Proof of Correctness for Algorithm $SSDST$ ............................................ 85
   The Unfair Distributed Daemon ................................................................ 90
   Self-Stabilizing Local Mutual Exclusion Algorithm on Rooted Trees $LMET$ 91
   Proof of Correctness of Algorithm $LMET$ ............................................. 92

CHAPTER 8  HEAP CONSTRUCTION .............................................................. 93
   Self-Stabilizing Min-Heap Algorithms for a Rooted Tree ....................... 93
   Heap Construction in a Rooted Tree ........................................................ 93
   Heap Construction in the Shared-Memory Model ..................................... 95
   Proof of Correctness of Algorithm $A\cdot HEAP$ ......................................... 96
   Reduction of Algorithm $A\cdot HEAP$ to $A\cdot HEAP$ ................................ 103
   Snap-Stabilizing Max-Heap Algorithm .................................................... 109
   Variables, Constants, and Configurations .............................................. 109
   Algorithm $Heap$ .................................................................................... 110
   Proof of Correctness of Algorithm $Heap$ .............................................. 119

CHAPTER 9  BINARY-SEARCH-TREE CONSTRUCTION .......................... 122
   BST Problem Specification ...................................................................... 122
   Binary-Search-Tree Algorithm ................................................................. 123
   Algorithm $BST$ .................................................................................... 124
   Proof of Correctness for Algorithm $BST$ .............................................. 133
   Abnormal Configurations and Delay ....................................................... 135
   Proof of correctness for Algorithm $BST$ .............................................. 136
   Unfair Daemon ...................................................................................... 140

CHAPTER 10  CONCLUSIONS ................................................................. 141

BIBLIOGRAPHY ...................................................................................... 143

VITA .......................................................................................................... 147

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF FIGURES

2.1 Chain orientation in the link-register model .............................................................. 19
2.2 $dist$ values for even and odd length chains ................................................................. 20
2.3 Ring orientation in the link-register model ................................................................. 26
2.4 $dist$ values for a ring with 9 nodes ............................................................................... 27

3.1 Four steps in a synchronous system ............................................................................... 31
3.2 Configuration $BAABAABABBBABA$ of a 16-node network ........................................... 35

4.1 Nodes $v$ and $r_v$ swap their IV values ........................................................................... 47
4.2 Pseudo-time values ............................................................................................................. 48

5.1 Execution steps in a synchronous system ........................................................................ 62
5.2 Example of a difference-string ......................................................................................... 64
5.3 Configuration $1#BAABAABABBBABA#0$ ......................................................................... 67

6.1 Pseudo-time values ............................................................................................................. 79

7.1 Four steps in a synchronous system ............................................................................... 85
7.2 Example of a tree with eight nodes ............................................................................... 86
7.3 Some tree with 9 nodes ................................................................................................... 89

8.1 Nodes $v$ and $J$ swap their IV values ........................................................................... 95
8.2 Pseudo-time values ............................................................................................................. 97
8.3 The seven states used by the algorithm ......................................................................... 110
8.4 Initial stage of constructing the heap. ............................................................................... 114
8.5 $M$ wave executed bottom-up in the 11-node tree ......................................................... 115
8.6 Macro $set.HV$s executed at node with $V = 100$ ......................................................... 116
8.7 Macro $set.HV$s executed at the node with $V = 145$ ................................................ 117
8.8 Macro $set.HV$s executed at node with $V = 130$ ........................................................ 117
8.9 The root and its children are done executing Module Heap. ....................................... 118
8.10 All the nodes are in $P$ state .......................................................................................... 118

9.1 Building a BST from a simple binary tree ................................................................. 123
9.2 The seven states used by the algorithm ....................................................................... 124
9.3 Some state ending the execution of Algorithm $Heap$ by the root and its children ....... 132
9.4 The tree is a BST ............................................................................................................. 133
ACKNOWLEDGMENTS

I would like to thank Dr. Ajoy Datta for chairing my committee and advising this work. I would like to thank Dr. Lawrence Larmore for continuously guiding and helping me to deepen my work. Without his generous help this dissertation would not have had such a rich content. For this and for being generous with their time when I needed it I am deeply indebted to them.

I would also like to specifically thank Dr. Vincent Villain, Dr. Laxmi Gewali, Dr. John Minor, and Dr. Shahram Latifi for serving on the committee. I would like to thank the faculty of the School of Computer Science of the University of Nevada, Las Vegas for the formal education along with generous financial support.

I am grateful to my parents for their inspiration. Special thanks go to my brother, Vlad and his wife Alina for their encouragement, and for the joy their son, Robert Andrei, gives to me. Finally and most importantly, I thank my husband, Wolfgang, for his love and support.
CHAPTER 1

INTRODUCTION

In this chapter we present some notions related to distributed systems (Section 1.1), topology and communication models (Section 1.2.1), distributed protocols and types of daemons (Section 1.3), self and snap-stabilization as particular cases of fault-tolerance (Section 1.4). We continue then with existent work in the literature (Section 1.5), our contributions (Section 1.6), and the organization of the dissertation (Section 1.7).

1.1 Distributed Systems

A recent orientation in computer systems research is to consider distributing computation among several processors (multiprocessor systems). If the processors share the computer bus, the clock, and sometimes memory and peripheral devices, the systems are called tightly coupled. If the processors do not share memory or clock, and instead have their own memory, they are called distributed systems. The processors communicate with each other either by using messages (message-passing communication model) or by using a common memory partitioned among processes such that each process has complete access over some portion, and limited access over the memory portions of other processes (shared-memory model). The two communication models are equivalent: A protocol written using one model can be rewritten using the other model. We have written our protocols mainly in the shared-memory model of communication.

There are various reasons for building a distributed algorithm using distributed systems: computation speed-up, reliability, communication. If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then a distributed system may allow us to distribute the computation among the various sites - to run that
computation concurrently. If one node fails in a distributed system, the remaining ones can potentially continue operating.

Regarding the timing of events (receiving/delivering a message, computing local information), we have several models of distributed systems.

- The **synchronous model** is the simplest model to describe and to program. We assume that all processors take steps in their executions simultaneously, and the transmission time of each message is bounded. But this is very difficult to implement, and because of this, most distributed systems are not synchronous.

- The **asynchronous model** is the other extreme, where processors can take steps at arbitrary speeds and in arbitrary orders. It is the hardest to program, because of the uncertainty in the order of events. Since the asynchronous model has no assumption about time, algorithms designed for the asynchronous model are general and portable: they are guaranteed to run correctly in networks with arbitrary timing guarantees. On the other hand, the asynchronous model does not provide sufficient conditions to solve problems efficiently, or even to solve them at all.

- The **partially synchronous model** is in between, with a wide range of possible assumptions that can be made. A very common assumption is to bound the interval of time for transmitting a message, called *timeout*, after which the message is considered lost. In our protocols we consider asynchronous systems, since they are the hardest to design for.

### 1.2 Topology and Communication Models

In this section we present some topology and communications models used by our algorithms.

#### 1.2.1 Topology Models

Regarding the topological structure of a distributed system, let $G = (V, E)$ be the underlying graph (directed or undirected). Let $E(v)$ denote the set of edged incident to node $v \in V$. Given $G$ and a set of labels $\Sigma$, a **local orientation** of $v \in V$ is any injective function $\lambda_v : E(v) \to \Sigma$ which associates a distinct label to each edge. The set of all the local orientations of the nodes in the graph, $\lambda = \{ \lambda_v : v \in V \}$, is called a **local orientation**.
labeling or simply labeling.

If any node $v$ can know from its local label what is the label at the other end of each of its incident edges, or formally, if there exists a bijection $\Psi : \Sigma \to \Sigma$ such that for each edge $e = < u, v > \in E$, $\lambda_v(< v, u >) = \Psi(\lambda_u(< u, v >))$, then the graph has edge-symmetry and $\Psi$ is called the edge-symmetry function [FKK+04].

If the symmetric function $\Psi$ is the identity function (for each edge, the labels at the two ends are the same) then the labeling function is called coloring. A particular case of coloring is the dimensional labeling in hypercubes.

By abuse of notation, a graph that has local labeling and edge symmetry is called an oriented graph. A network with local labeling but no edge symmetry is called unoriented [FKK+04]. Examples of oriented graphs are: “left-right” labeling for chains and rings, “north-south/east-west” labeling for meshes and tori, “parent-child” labeling for trees.

We consider three topologies: an asynchronous chain of $n$ nodes, an asynchronous ring network of $n$ nodes with one distinguished node, which we call the leader, and an asynchronous rooted tree of $n$ nodes and height $h$, with one distinguished node called the root. Chains, rings and trees are assumed to be bidirectional and oriented.

In an unoriented chain or ring, a node may have either one or two neighbors, called first ($f$) and second ($s$) neighbor, stored as constants. In an oriented tree, a node may have a non-empty set of neighbors, called $n_1$ (the first neighbor), $n_2$ (the second neighbor) etc., stored as constants. We call them constants since none of the proposed algorithms modify them (so call static storage), but they are not constants for the system, since transient failures can modify them. If some neighbor is missing (for extremity nodes), then it is stored as $\bot$.

For an oriented chain and ring, every node $v$ can distinguish between its left neighbor ($l_v$) and right neighbor ($r_v$), and this left-right orientation must be consistent among all the nodes in the network. If node $v$ does not have one of the two neighbors, the corresponding value is represented as $\bot$. For the chain, the leftmost node is denoted by $L$, the rightmost node by $R$; for the ring, the ring leader is denoted by $L$. We assume that an underlying self-stabilizing local maintenance protocol maintains the left neighbor pointer $l_v$ and the right neighbor pointer $r_v$ of a node $v$.

For a tree, every node $v$ can distinguish between its parent ($p_v$) and its children (set $D_v$).
The root node is denoted by $R$. We assume that an underlying self-stabilizing spanning tree construction protocol maintains the parent pointer $p_v$ and set of neighbors $N_v$ of a node $v$. All neighbors of $v$ except its parent $p_v$ are considered to be its children, denoted as the set $D_v$. Note that $D_v$ is not maintained by any protocol as it can be locally computed from $N_v$ and $p_v$. Thus, every node $v$ effectively maintains $deg_v$ pointers, where $deg_v$ is the degree of node $v$. For the root node $R$, $p_R = \perp$. For a leaf node $v$, $D_v = \perp$.

1.2.2 Communication Models

Among the several models for interprocess communication, two communication models are used by our algorithms. In the state-reading model [Dij74], also called *shared-memory* model, processors communicate by reading the neighbors' state. In the link-register model [DIM93], a process uses separate shared-registers to communicate with its neighbors.

In the shared-memory model, a process can read and write its own memory, but can only read the memory of its neighbors.

In the link-register model, processes $u$ and $v$ communicate using two separate registers: $S_{uv}$ and $S_{vu}$. Process $u$ writes in $S_{uv}$ and reads from $S_{vu}$, whereas process $v$ writes in $S_{vu}$ and reads from $S_{uv}$. In the shared-memory model, process $u$ writes in register $S_u$ readable by both $u$ and $v$.

We use mainly shared-memory model, since it is harder to design an algorithm in the shared-memory model than in the link-register model.

1.3 Distributed Protocols and Types of Daemons

A distributed program (protocol) is executed by individual processes. We use the term process and node interchangeably. Each process holds a number of variables (and sometimes, constants). The variables in the shared-memory model can be of two types: *private* or *local* variables, that are visible only by the process, and *shared* or *global* variables, that can be visible by other processes in the system.

The *local state* is the set of values for the process' variables. The *global state* (configuration) of a system is the union of the local state of its components.

The distributed program in every node consists of a finite set of guarded actions of the
The guard of an action is a Boolean expression involving the global variables and/or local variables. The action can be executed only if its guard evaluates to true. A node with at least one enabled guard is called enabled. A daemon will select non-deterministically a non-empty subset of enabled nodes to execute one of the enabled actions.

We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding action, if it is selected for execution, are done in one atomic step.

[DIM93] distinguishes between composite and read/write atomicity protocols. The guards of a read/write atomicity protocol are only of two types:

1. either the guard is defined only over local variables, and the action involves global and/or local variables,
2. or the guard is defined over local and/or global variables, and the action involves only local variables.

The guards of a composite atomicity protocol can also include a third type:

3. the guard is defined over local and/or global variables, while the action can involve both local and global variables.

In the system, one or more processors execute an action and a processor may take at most one action. This execution model is known as a daemon.

- The central daemon selects only one enabled process for execution.
- The distributed daemon selects a non-empty set of enabled processes for execution.

There are several types of distributed daemons. The most common are:

- The weakly fair daemon is a distributed daemon that assumes fairness: a continuously enabled process will be eventually selected for execution.
- The unfair daemon is a distributed daemon without the fairness mechanism: even a continuously enabled process may not be selected for execution unless it is the only enabled process.
In order to compute the time complexity for an algorithm running on an asynchronous system, we use the definition of a *round* [DIM97]. A round is a minimal sequence of computation steps during which each processor that was enabled in the first configuration of the sequence executes at least once during this sequence.

1.4 Self-Stabilization and Snap-Stabilization

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 has occurred. A $k$ fault-tolerant system can tolerate up to $k$ faults, provided that the function the system has to perform is independent on the number of processes not corrupted in the system. It may or may not tolerate more than $k$ faults, but the guarantee is only for at most $k$ faults.

In 1973, Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [Dij74]. He 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”. A system which is not self-stabilizing may stay in an illegitimate state forever.

Given $C$, the set of all possible states, and a predicate $P$ over $C$, we denote $\mathcal{L}_P \subseteq C$ the set of all legitimate states with respect to $P$, or simply the set of all legitimate states.

**Definition 1.4.1 (Closed Attractor)** Let $C_1$ and $C_2$ be subsets of $C$. $C_2$ is a closed attractor for $C_1$ if the following conditions are true:

(i) for any initial state $c_i$ in $C_1$, for any execution $e$ in $E_{c_i}$ ($e = c_1, c_2, \ldots$), 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 1.4.2 (Self-Stabilization)** A system $S$ is called self-stabilizing if and there exists a predicate $P$ such that $\mathcal{L}_P$ is a closed attractor for $C$.

A *snap-stabilizing* algorithm [BDPV99, CDPV03] guarantees that the system always behaves according to its specification provided some processor initiated the protocol. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes...
in 0 steps. It should be noted that a self-stabilizing algorithm is guaranteed to satisfy the desired specification only in a finite time.

We assume that in a normal execution, at least one processor (called the initiator) initiates the protocol upon an external (with respect to the protocol) request by executing a special type of action, called an initialization action.

**Definition 1.4.3 (Snap-Stabilization)** Let $P$ be a protocol designed to solve a task $T$. $P$ is called snap-stabilizing if and only if, starting from any configuration, any execution $E$ of $P$ always satisfies the specification of $T$.

### 1.5 Related Work

Distributed algorithms aim to achieve better performance than sequential algorithms in terms of time complexity (or asymptotic time complexity) while keeping or lowering the memory requirement (space complexity) in a node. (In sequential algorithms, the memory requirement is the memory requirement of the algorithm itself.)

Self-stabilizing distributed algorithms aim to achieve comparable performance to non-stabilizing distributed algorithms when transient faults or arbitrary initialization cause the system to enter a state where a non-stabilizing algorithm cannot continue to properly perform its task.

Transient faults can affect an existing data structure and alter its data content. As a result, the data structure may lose its properties, and the operations defined over the data structure will have unpredictable and undesirable results, making the data structure unusable.

#### 1.5.1 Self-Stabilizing Distributed Sorting

There are various types of the sorting problem for distributed systems.

One sorting problem in a general (unreliable) network where nodes have distinct IDs can be formulated as follows: Given a set of $n$ values and $n$ nodes, distribute the values among the nodes in the increasing order of the node ID. We call this type of sorting as type UI (unreliable network, unique IDs).
Another type of sorting problem in a general reliable network where distinct IDs for the nodes are not considered can be formulated as follows: Given \( n \) values and \( n \) nodes arranged as an ordered type of (di)graph (we mean a line, a ring with a distinguish node, or a tree), distribute the values among the nodes in the increasing order of the node topological position. For example, in a line network, the order is from left to right, or from right to left. For a ring network, the order is from the distinguish node to the right, or to the left. For a tree network, the order will be from the root to the leaves, or from the leaves to the root.

We call this type of sorting as type RA (reliable network, anonymous nodes).

Type UI has been studied in [ABCD96, ABC+98, BDT98, GZ97].

In [ABCD96], a global self-stabilizing distributed algorithm for the sorting problem in a tree network where nodes have distinct IDs is proposed. Since the protocol maintains in every node a consistent view of the system state, it has few drawbacks. The number of nodes \( n \) needs to be known by every node. The stabilization time is \( O(d) \), where \( d \) is the network diameter, instead of \( O(h) \), where \( h \) is the height of the tree. The memory requirement in every node is of \( O(n) \), which makes a total of \( O(n^2) \) for the whole network.

The memory requirement per node of the solution from [ABCD96] is decreased to \( O(\log(n)\text{deg}) \) in [ABC+98], where \( \text{deg} \) is the degree of the tree, but with a cost of a much higher time complexity \( O(nh) \).

In [BDT98], two solutions based on message passing are given for the sorting problem in a unidirectional ring (one uses the store-and-forward routing, the other uses cut-through routing). The size of a message is \( O(n \log n) \). The memory requirement of a node is either \( O(n \log n) \) or \( O(\log n) \) (depending on the routing scheme). The stabilization time is \( O(n^3 \log n) \), respectively \( O(n^2 \log n) \).

In [GZ97], the upper and lower bounds on the maximal number of bits sent during the execution of an algorithm in an asynchronous network with \( n \) nodes are studied. For a tree network, the lower bound on the number of bits is \( O(\Delta_T \log L/n) \), where \( \Delta_T \) is the sum of the distances from all the vertices to the median of the tree, and \( \{1, 2, \ldots, L\} \) is the set of initial values. An algorithm that sends at most \( O(\Delta_T \log Ln/\Delta_T) \) bits is presented (upper bound). The upper and the lower bounds are tight if either \( L = \Omega(n^{1+\varepsilon}) \) or \( \Delta_T = \Omega(n^2) \).

Some results regarding average distribution are also presented, and these results suggest...
that sorting is an inherently non-distributive problem.

Type RA has been studied in [Sas02, Sas04, FKK+04]. For a non fault-tolerant perspective, the main concern is to reduce the amount of communication in a message-passing model, or the amount of memory accesses (comparisons) for a shared-memory model, while keeping a relative low memory requirement per node, and a low time complexity.

We consider the type RA of the sorting problem, in a asynchronous line network. Given \( n \) values and \( n \) nodes arranged as a line, distribute the values among the nodes in the increasing order of the node position from left to right. For non-descending order, distribute the values in non-descending order from left to right (similarly for non-ascending order). The values are not necessarily distinct, and they are drawn from an arbitrary set of values.

The type RA sorting problem becomes a heap problem when the network is a rooted tree (either min heap or max heap). Given \( n \) values and \( n \) nodes arranged as a rooted tree, distribute the values among the nodes in the increasing order of their depths.

A time optimal solution to the type RA sorting problem in a synchronous line network is given in [Sas02]. The strict lower bound of \( n - 1 \) rounds is achieved by creating copies of the elements in each node. The space complexity in every node is \( O(L) \).

A time and communication optimal solution to the classical sorting problem in an asynchronous line network is given in [Sas04]. The time complexity is \( n - 1 \), thus optimal, and the communication complexity is \( n^2/2 \). The algorithm uses three states and \( O(L) \) space complexity per node. Even if the algorithm works for asynchronous systems, it is completely non-fault tolerant. It assumes a correct initialization, and if a fault occurs, it runs forever.

An interesting analysis of the relationship between sorting and election in an anonymous asynchronous ring is done in [FKK+04]. For the case when the input values are drawn from the set \{0,1\} and the size \( n \) of the ring is prime, a lower bound on the message complexity for both oriented and unoriented rings is given, together with a relatively tight upper bound achieved by the given algorithms. For oriented rings, the number of states is 11. The space complexity for both oriented and unoriented ring algorithms is \( O(\log n) \), as every node maintains a counter that takes values in the set \( 1 \ldots n \).

Our sorting algorithms for both the line and the tree networks use a so-called local resource allocation (LRA) protocol [CDP03] to deal with the exchange of two values among
neighboring nodes. LRA allows neighboring processes to access resources (i.e. their values to be sorted) concurrently, provided they are not conflicting with each other. LRA protocol becomes, for a line network, a local mutual exclusion protocol, where when a node is enabled, its direct neighbors are disabled. LRA protocol is not a mutual exclusion protocol in a ring network, provided that if two neighboring nodes $u$ and $v$ are enabled at the same time, the execution of node $u$ and node $v$ involves disjoint (non-conflicting) resources.

1.5.2 Self-Stabilizing Heap and Binary-Search-Tree

The self-stabilizing heap problem has been studied in [BD95, Ali99, HM01a, HM01b, UHK+03].

In [Ali99], the self-stabilizing algorithm for a max-heap construction improves the version of [BD95] in three aspects. The algorithm uses the shared-memory model of communication. First, no global reset is required compared with [BD95]. Second, the time complexity is reduced from $O(nh)$ to $O(h)$, where $h$ is the height of the tree and $n$ is the number of nodes in the tree. Finally, the space complexity per node is reduced from $O(degL)$ to $O(deg + L)$ (where $deg$ is the degree of the process and $L$ is the maximum size of the initial values in the tree). Since the memory requirement in a node is independent of the number of nodes, it is assumed to be $O(1)$. The synchronization among the nodes is achieved by using the global rooted synchronizer presented in [ABDT98] and two additional bits.

In [UHK+03], the self-stabilizing max-heap protocol reduces the memory requirement further to $O(L)$, by keeping the same order for the time complexity ($O(h)$). It uses a neighborhood synchronizer protocol [JADT99]. Besides the two variables of $O(L)$ used for temporarily storing the values to be exchanged between the node and its child, three more bits are used per node as follows: one bit for synchronization, one bit for marking changes, and one bit for resetting the heap construction in the subtree rooted at the node.

A heap construction that supports insert and delete operations in arbitrary states over a variant of the standard binary heap [CLR92] with the maximum capacity of $K$ items is proposed in [HM01a]. It takes $O(m \log K)$ heap operations to stabilize ($m$ is the initial number of items in the heap). The space complexity per node $i$ is $O(h_i)$ where $h_i$ is the height of the subtree $T_i$ in the binary heap rooted at node $i$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Stabilizing search 2-3 trees are investigated in [HM01b]. The stabilization time is \( O(n \log n) \) rounds where \( n \) is the number of nodes in the initial state and the space complexity per node \( i \) is \( O(d_i) \) where \( d_i \) is the distance from the root to node \( i \).

1.5.3 Self-Stabilizing Local Mutual Exclusion

Local mutual exclusion (LME) is the extension of the dining philosophers problem [Dij71] to a general network. It weakens the mutual exclusion by requiring at most one privileged node per neighborhood, but at least one privileged node in the network.

Awerbuch et. al. [AS90] proposed a polynomial response time solution to the dining philosopher problem. Chandy et. al. [CM84] proposed a solution to an alternative local mutual exclusion problem called drinking philosopher problem. Unfortunately their solutions does not handle transient faults (are not self-stabilizing, just distributed).

Hoover and Poole [HP89] and Gouda [Gou87] proposed self-stabilizing solutions to the dining philosophers problem that use a central daemon and a distinguish process to implement the token circulation.

Antonoiu et. al [AS98, AS99] proposed two self-stabilizing LME algorithms. The protocol of [AS98] works on oriented trees (each node knows its parent and its set of children), with 4 states per process. The protocol of [AS99] works on general networks where the IDs of the nodes are locally unique, with \( O(M) \) states per process, where \( M \) is an integer much larger than \( n \) (\( M >> n \)).

Gouda and Haddix [GH99] solution to the LME problem used variables bounded by \( 2d \) where \( d \) is the length of the longest simple cycle in the network. Nesterenko and Arora [AN02] solution to the dining philosopher problem (LME on a ring) uses read/write atomicity and variables bounded by 4.

The closest to our work is the algorithm of [Hua00]. Huang [Hua00] proposed two algorithms for synchronous systems. One algorithm uses only two states per process, but assumes no transient faults. The second protocol handles transient faults but more states are added (the network is initially colored such that no two adjacent links use the same color). At the same time, the delay between two consecutive executions of the same process is at most \( n \) and never is reduced.
1.5.4 Orientation of a Ring

Up to our knowledge, there is no work on orienting a chain. For ring orientation, extensive work has been done in [ASW88, SP87, BP89, ILS95, IJ93, TH95] Attiya et. al. [ASW88] focused on uniform, anonymous, asynchronous rings, and showed that there is no deterministic protocol to orient even-length anonymous rings, and there is no protocol to orient arbitrary length rings that terminates [SP87]. Syrotiuk and Pachl [SP87] presented a simple asynchronous protocol to orient odd and bounded rings, using message-passing model. The protocol is not fault-tolerant. Burns and Pachl [BP89] showed that deterministic self-stabilizing protocols can break symmetry on oriented rings of prime size, and Itkis et. al. [ILS95] presented a constant space solution to this case.

Some non-existence results have been given by Israeli and Jalfon [IJ93]. Also, in [IJ93], a randomized self-stabilizing ring orientation protocol for arbitrary rings and a uniform deterministic self-stabilizing odd-length-ring orientation protocol in the link-register model [DIM93] are also given. Tsai and Huang [TH95] present a deterministic protocol for arbitrary rings, under the central daemon, when the neighbor of a node has more knowledge regarding the edge orientation. Finally, Hoepman [Hoe98] presents two uniform deterministic self-stabilizing ring-orientation protocols for odd-length rings, using constant space, one using the link-register model, the other using the state-reading model.

1.6 Contributions

Our contribution is diverse.

To orient an unoriented chain, we propose two uniform algorithms. Algorithm $SEN_{\text{chain}}$, implemented in link-register model, requires that the extremity nodes have distinct IDs. The space complexity in a node is independent of $n$, and the stabilization time is at most $n$ rounds. Algorithm $SEN_{\text{chain}}$, implemented in the shared-memory model, requires that the nodes' IDs are at least chromatic [KY02] (no two processes neighboring to each other have the same identifiers). It uses $O(\log n)$ bits per node and stabilizes in at most $n$ rounds.

We propose a space and time optimal self-stabilizing distributed algorithm for simultaneously activating non-adjacent processes on an oriented chain (Algorithm $SSDS$).
We use Algorithm \textit{SSDS}, to accomplish two tasks: local mutual exclusion and line sorting. We propose two uniform, self-stabilizing, deterministic protocols on oriented chains: a time and space optimal solution to the local mutual exclusion problem (Algorithm \textit{LMEC}), and a space and (asymptotic) time optimal solution to the distributed sorting problem (Algorithm \textit{SORT}_c).

Algorithm \textit{LMEC} uses only two states and stabilizes in 0 rounds (it is \textit{snap}-stabilizing). The space requirement per node is three bits. Thus by combining algorithms \textit{LMEC} and either \textit{SENS}_1^{\text{chain}} or \textit{SENS}_2^{\text{chain}}, we obtain a solution to the local mutual exclusion on an unoriented chain, with \(O(1)\) bits (respectively \(O(\log n)\) bits) memory size and \(n\) rounds as upper bound on the stabilization time.

Algorithm \textit{SORT}_c sorts \(n\) values, not necessarily distinct, in non-decreasing order from left to right. Each process holds only one value, at any moment. It uses a total of three bits per node, thus an improvement over [Sas02, Sas04]: except the value to be sorted, only three bits of memory are used in each node. It stabilizes in at most \(8n - 8\) rounds, comparing with \(n - 1\) rounds achieved by non-stabilizing algorithms in [Sas02, Sas04]. Thus by combining algorithms \textit{SORT}_c and either \textit{SENS}_1^{\text{chain}} or \textit{SENS}_2^{\text{chain}}, we obtain a solution to the sorting problem on an unoriented chain, with \(O(1)\) bits (respectively \(O(\log n)\) bits) memory size and and \(9n - 8\) rounds as upper bound on the stabilization time.

We then give two solutions to the ring orientation. Algorithm \textit{SENS}_1^{\text{ring}} is implemented in link-register model, uses \(O(1)\) bits per node, and stabilizes in at most \(n\) rounds. Algorithm \textit{SENS}_2^{\text{ring}} is implemented in shared-memory model, uses \(O(\log n)\) bits per node, and stabilizes in at most \(n\) rounds.

We extend Algorithm \textit{SSDS} to an asynchronous oriented ring with a distinguished node with some minor modifications, and we obtain general self-stabilization for simultaneously activated non-adjacent processes in an oriented ring with a distinguished process (Algorithm \textit{SSDSR}).

Algorithm \textit{SSDSR} is deterministic, and (asymptotically) optimal in time and space. It is semi-uniform and works under the weakly fair daemon but not under the unfair distributed daemon. It uses at most two bits per node (\(n - 2\) nodes use one bit each and the other two nodes use two bits each, where \(n\) is the number of nodes in the network). It is asymptotically
optimal in the time complexity — for any $t \geq 0$, every node is enabled at least $t$ times within
the first $3n/2 - 2 + 3t$ rounds, i.e., on the average, once every three rounds.

We use Algorithm $SSDSR$, to accomplish two tasks: local resource allocation and ring
sorting. We propose two uniform, self-stabilizing, deterministic protocols on oriented rings: a
time and space optimal solution to the local resource allocation problem (Algorithm $LRAR$),
and a space and (asymptotic) time optimal solution to the distributed sorting problem
(Algorithm $SORT_r$).

Algorithm $LRAR$ uses at most four states ($n - 2$ nodes use two states each and the other
two nodes use four states each), and stabilizes in 0 rounds (it is snap-stabilizing). The space
requirement per node is $\log n$ bits. Thus by combining algorithms $LRAR$ and either $SENS_1^{ring}$
or $SENS_2^{ring}$, we obtain a solution to the local resource allocation on an unoriented ring,
with $O(1)$ bits (respectively $O(\log n)$ bits) memory size and $n$ rounds as upper bound on
the stabilization time.

Algorithm $SORT_r$ sorts $n$ values on a ring in non-decreasing order from left to right
starting from the distinguished node in at most $4(5n - 6)$ rounds. It uses at most four bits
per node. (Two distinguished nodes use four bits each, while the other nodes use three
bits each.) Thus by combining algorithms $SORT_c$ and either $SENS_1^{ring}$ or $SENS_2^{ring}$, we
obtain a solution to the sorting problem on an unoriented ring, with $O(1)$ bits (respectively
$O(\log n)$ bits) memory size and and $n + 4(5n - 6)$ rounds as upper bound on the stabilization
time.

We extend Algorithm $SSDS$ to an asynchronous rooted tree, and we obtain general self-
stabilization for simultaneously activated non-adjacent processes in a rooted tree (Algorithm
$SSDST$).

Algorithm $SSDST$ is deterministic, space and (asymptotically) time optimal. It is
uniform and works under any unfair distributed daemon. It uses $\lceil \log(deg) \rceil$ bits per node
(where $deg$ is degree of the node), and stabilizes in at most $2h + 2t - 1$ rounds (where $h$ is
the height of the tree) to the global predicate: every node has executed its application at
least $t$ times, $t \geq 0$.

We then give two applications of Algorithm $SSDST$: a time and space optimal solution
to the local mutual exclusion problem (Algorithm $LMET$) and a space and (asymptotically)
time optimal solution to the min heap problem (Algorithm \texttt{HEAP}).

Algorithm \texttt{LMET} uses only one bit per node and stabilizes in 0 rounds (it is \textit{snap-stabilizing}). It is 1-fair and during the first $2h + 2t - 1$ rounds, a node enters its CS at least $t$ times, i.e., on the average, once every two rounds.

Algorithm \texttt{HEAP} arranges $n$ values not necessarily distinct, in non-decreasing order from top to bottom (min heap), in at most $4(7h/2 - 4)$ rounds. Each process holds only one value at any moment. It uses a total of $[\log(deg)]$ bits ($deg$ is the node degree) which is optimal, thus an improvement over [UHK+03, BDVQ05]: except the value to be sorted, only $2 + [\log(deg)]$ bits of memory are used in each node. The time complexity is $4(7h/2 - 4)$ rounds, which is asymptotically optimal.

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

In proving the time complexity of sorting, we introduce the notion of \textit{pseudo-time}, similar to \textit{logical time} introduced by Lamport [Lam78]. Each node in the network has a “local clock” which has the property that when a certain comparison must be executed between the node and its right neighbor, both nodes have the same value of the local clock.

We present the first snap-stabilizing distributed \textit{binary search tree} (BST) algorithm. The maximum number of items that can be stored at any time at any node is independent of the size $n$ of the network. Under this space constraint, we show a lower bound of $\Omega(n)$ on the time complexity for the BST problem. We then prove that starting from an arbitrary configuration where the nodes have distinct internal values drawn from an arbitrary set, our algorithm arranges them in a BST order in $O(n)$ rounds. Therefore, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor $i$ requires $O(\log s_i)$ bits of space where $s_i$ is the size of the subtree rooted at $i$. So, the root uses $O(\log n)$ bits.

The proposed algorithm uses a \textit{heap} algorithm (Algorithm \texttt{Heap}) as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem. The heap construction spends $O(h)$ (where $h$ is the height of the tree) rounds. Its space requirement is constant (independent of $n$). We then exploit the heap in the next phase of the protocol.
The root collects values in decreasing order and delivers them to each node in the tree in \( O(n) \) rounds following a pipelined delivery order of sorted values in decreasing order.

1.7 Organization of Dissertation

Chapter 2 contains the communication and topology models used in our algorithms, and some proposed algorithms for orienting a chain and a ring.

Chapter 3 contains Algorithms \( SSDS \) and \( LMEC \) that are self-stabilizing algorithms on a chain network.

Chapter 4 contains Algorithm \( ASORT_c \) and \( SORT_c \) that are self-stabilizing sorting algorithm on an oriented chain network.

Chapter 5 contains Algorithms \( SSDSR \) and \( LRAR \) that are self-stabilizing algorithms on a ring network.

Chapter 6 contains Algorithm \( ASORT_r \) and \( SORT_r \) that are self-stabilizing sorting algorithm on an oriented ring network.

Chapter 7 contains Algorithms \( SS DST \) and \( LMET \) that are self-stabilizing algorithms on a tree network.

Chapter 8 contains Algorithm \( ASH2AP \) and \( H2AP \) that are self-stabilizing min-heap algorithms, and Algorithm \( Heap \) that is a snap-stabilizing max-heap algorithm.

Chapter 9 contains Algorithm \( BST \) that is a snap-stabilizing algorithm for constructing a binary-search-tree on a tree network.

We finish with concluding remarks and future work on Chapter 10.
CHAPTER 2

PRELIMINARIES

In this chapter we present another particular type of shared-memory model, a so-called *abstract* model (Section 2.1). We then define what means to reduce an algorithm in some model to another algorithm in another model, both algorithms solving the same problem.

Since our algorithms work on oriented topologies (oriented chains, oriented rings, rooted and oriented trees), we then present in Section 2.2 two algorithms for orienting a chain network and in Section 2.3 two algorithms for orienting a ring network (their proof of correctness is also included).

2.1 Reducing an Algorithm in Abstract Model to an Algorithm in Shared-Memory Model

In this section we define what the abstract model of communication is and what means that two algorithms solving the same problem can be reduced one to another: an algorithm written in some model can be reduced to another algorithm written in another model, specifically from the abstract model to the shared-memory model.

In the *abstract* model of communication, different from the shared-memory model, in which an enabled node \( v \) can modify a single variable of some neighbor \( w \), if the local mutual exclusion is satisfied locally. Specifically, we allow a node to change the variable \( IV \) of some neighboring node \( w \) in order to perform the swapping of the two values (a node can then synchronize the swap of values with some node \( w \)). We then assume that the swap is done in one atomic step, and we show later how this is done in the shared-memory model.

**Definition 2.1.1 (Reduction)** *Given two different models of communication \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \), an algorithm \( A_1 \) in the model \( \mathcal{M}_1 \) can be reduced to another algorithm \( A_2 \) in the model \( \mathcal{M}_2 \) 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 \( A_2 \) accomplishes a task in the model \( M_1 \) and \( A_1 \) reduces to \( A_2 \), then by Definition 2.1.1, \( A_1 \) accomplishes the same task in the model \( M_2 \).

2.2 Algorithms for Orienting an Unoriented Chain

In this section we propose two algorithms for orienting an unoriented chain with at least two nodes \( (n \geq 2) \), followed by their proof of correctness in Section 2.2.1.

Algorithm \( SENS_{1}^{chain} \), implemented in the link-register model, requires that the IDs of the extremity nodes to be distinct. Algorithm \( SENS_{2}^{chain} \), implemented in shared-memory model, requires that the nodes' IDs are at least chromatic [KY02]: The process identifiers are said to be chromatic if no two processes neighboring to each other have the same identifiers.

The purpose of algorithms \( SENS_{1}^{chain} \) and \( SENS_{2}^{chain} \) is to compute for each node \( v \) the directions \( left.v, right.v \in N_v \) such that \( \left( left.v = \bot \lor right.(left.v) = v \right) \land \left( right.v = \bot \lor left.(right.v) = v \right) \).

Algorithm \( SENS_{1}^{chain} \) (Figure 2.2) works as follows. The two extremity nodes have distinct IDs, so one extremity has a lesser value (\( minID \)) than the other (\( maxID \)). Each extremity write in the (only) register its ID. An internal process reads the registers from its neighbors and forwards the content. A node decides on its orientation based on the current two values held in the read-registers: it consider its left as the node from which it reads the smallest value (that will eventually be \( minID \)) and as its right as the node from which it
reads the largest value (that will eventually be $\text{maxID}$) (see Figure 2.1).

A node $v$ will write into registers $R_f$, $R_s$ what it is has to communicate to the first, respectively second neighbor. Also, node $v$ will read from registers $R'_f$, $R'_s$ what the first, respectively second neighbor has sent to it.

Predicate $\text{consistent}(v)$ is true when node $v$ has its registers consistent with its neighbors' registers.

Algorithm $\text{SENS}_1^{\text{chain}}$ is uniform and does not terminate (Guards $O_l$ and $O_r$ are executed infinitely often). The space complexity in a node is independent of $n$ (depends only on the size of nodes' IDs). The stabilization time is at most $n$ rounds (Lemma 2.1).

Algorithm $\text{SENS}_2^{\text{chain}}$ works as follows. Every node in the chain has a certain distance to the closest extremity node in the chain (we call that distance $\text{dist}(v)$). For each extremity node $v$, $\text{dist}(v) = 0$. For some internal node $v$, $\text{dist}(v) = \min_{j \in \mathbb{N}_n} \text{dist}(j) + 1$.

We use $\text{dist}$ values to select the unique node in the chain (called decider) to decide the orientation of every node in the chain. In case $n$ is odd, the decider is the node in the middle. In case $n$ is even, the decider is the node with the higher ID among the two nodes in the middle. In Figure 2.2, even and odd-number chains are considered.

The decider has a double purpose. One purpose is to be the only one to start the decision process on the edges' orientation. The second purpose is to help a node in selecting the neighbor to follow in deciding the orientation of the edges adjacent to it (every other node will select the neighbor closer to the decider).
Algorithm 2.2.1 Self-Stabilizing Orientation of a Chain $SNS_{\text{chain}}$

**Predicate** $\text{consistent}(v) \equiv (|N_v| = 1 \land R_f = ID_v) \lor (|N_v| = 2 \land R_f = R_s' \land R_s = R_f')$

**Actions executed by the extremity nodes of the chain**

\begin{align*}
b & \quad |N_v| = 1 \land R_f \neq ID_v \implies R_f = ID_v \\
O_r & \quad |N_v| = 1 \land \text{consistent}(v) \land R_f' > ID_v \implies \\
& \quad \text{right}.v = f \\
& \quad \text{left}.v = \bot \\
O_l & \quad |N_v| = 1 \land \text{consistent}(v) \land R_f' < ID_v \implies \\
& \quad \text{left}.v = f \\
& \quad \text{right}.v = \bot
\end{align*}

**Actions executed by the internal nodes of the chain**

\begin{align*}
b & \quad |N_v| = 2 \land (R_f \neq R_s' \lor R_s \neq R_f') \implies \\
& \quad \text{if } (R_f \neq R_s') \text{ then } R_f = R_f' \\
& \quad \text{if } (R_s \neq R_f') \text{ then } R_s = R_f' \\
O_r & \quad |N_v| = 2 \land \text{consistent}(v) \land R_f' > R_s' \implies \\
& \quad \text{right}.v = f \\
& \quad \text{left}.v = s \\
O_l & \quad |N_v| = 2 \land \text{consistent}(v) \land R_f' < R_s' \implies \\
& \quad \text{left}.v = f \\
& \quad \text{right}.v = s
\end{align*}

---

(a) $dist$ values when $n = 10$  

(b) $dist$ values when $n = 11$

Figure 2.2: $dist$ values for even and odd length chains

Every node holds $v$ a variable $x.v$. The set of all the $x$ values for the entire chain can be considered as an array $x$ (instead of $x.v$ we can write $x[v]$). Let $X(v)$ be an abstract function of the physical values $x$ of the neighbors of node $v$ defined as follows:

(i) if $|N_v| = 1$ (if $v$ is an extremity of the chain), then $X(v) = 0$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
(ii) if $|N_v| = 2$, then

$$X(v) = \max \left\{ \frac{1}{\min_{j \in N_v} x[j]} + 1 \right\}$$

Starting from an arbitrary state, each node in the chain sets its $x$ value to $X(v)$ and eventually converges to $\text{dist}(v)$ in finite time.

The decider $v$ has the property that one and only one of the following two conditions is true:

(i) $(x.f = x.s)$ for odd-number (even-length chains), or

(ii) $(x.v = x.f \land ID.v > ID.f) \lor (x.v = x.s \land ID.v > ID.s)$ for even-number (odd length chain).

Algorithm $\mathcal{SENS}^{chain}_2$ is presented next (Figure 2.2). By abuse of notation, for the two neighbors $f$ and $s$ of some node $v$, $\neg f = s$ and $\neg s = f$.

Guard $X$ is responsible for making sure that $x.v = X(v)$.

One and only one of the guards $D_o$ and $D_e$ is executed by the decider node, responsible for deciding the orientation of the entire chain. When the decider has decided a correct orientation for itself (which is its left and which is its right neighbor), Predicate $\text{has_orientation}$ is evaluated to true. Guard $D_o$ is executed by the decider when it has no correct orientation and the chain has odd number of nodes. Guard $D_e$ is executed by the decider when it has no correct orientation and the chain has even number of nodes.

When a non-decider node has a correct orientation, Predicate $\text{is_oriented}$ is evaluated to true. Guards $O_l$ and $O_r$ are executed by a non-decider node when it has no correct orientation. Then it decides the orientation based on the orientation of one of its neighbors that is closer to the decider node (the $x$ value of that neighbor is higher or equal to the node's $x$ value).

Algorithm $\mathcal{SENS}^{chain}_2$ is uniform, deterministic, and eventually terminates. The stabilization time is at most $n$ rounds (Lemma 2.2).
Algorithm 2.2.2 *Self-Stabilizing Orientation of a Chain* $S\mathcal{E}N\mathcal{S}_{2}^{chain}$

**Predicates**

\[
\text{is\_decider}(v) \equiv (x.f = x.s) \lor (\exists i \in \{f, s\} : x.v = x.i \land ID.v > ID.i)
\]

\[
\text{is\_oriented}(v, v_{left}, v_{right}) \equiv \text{left}.v = v_{left} \land \text{right}.v = v_{right}
\]

\[
\text{has\_orientation}(v) \equiv (\text{left}.v = f \land \text{right}.v = s) \lor (\text{left}.v = s \land \text{right}.v = f)
\]

**Actions executed by the extremity nodes of the chain**

\[
X \quad |N_{0}| = 1 \land x.v \neq 0 \quad \rightarrow \quad x.v = 0
\]

\[
\text{left}.v = \text{right}.v = \bot
\]

\[
O_{1} \quad |N_{0}| = 1 \land x.v = 0 \land \neg \text{is\_decider}(v) \land x.f \geq x.v \land \text{left}.f = v
\]

\[
\neg \text{is\_oriented}(v, \bot, f) \quad \rightarrow
\]

\[
\text{right}.v = f
\]

\[
\text{left}.v = \bot
\]

\[
O_{2} \quad |N_{0}| = 1 \land x.v = 0 \land \neg \text{is\_decider}(v) \land x.f \geq x.v \land \text{right}.f = v
\]

\[
\neg \text{is\_oriented}(v, f, \bot) \quad \rightarrow
\]

\[
\text{left}.v = f
\]

\[
\text{right}.v = \bot
\]

**Actions executed by the internal nodes of the chain**

\[
X \quad |N_{0}| = 2 \land x.v \neq X(v) \quad \rightarrow
\]

\[
x.v = X(v)
\]

\[
\text{left}.v = \text{right}.v = \bot
\]

\[
D_{0} \quad |N_{0}| = 2 \land x.v = X(v) \land x.f = x.s \land \neg \text{has\_orientation}(v) \quad \rightarrow
\]

\[
\text{left}.v = f
\]

\[
\text{right}.v = \bot
\]

\[
D_{2} \quad |N_{0}| = 2 \land x.v = X(v) \land (\exists i \in \{f, s\} : x.v = x.i \land ID.v > ID.i) \land
\]

\[
\neg \text{has\_orientation}(v) \quad \rightarrow
\]

\[
\text{left}.v = i
\]

\[
\text{right}.v = \neg i
\]

\[
O_{1} \quad |N_{0}| = 2 \land x.v = X(v) \land \neg \text{is\_decider}(v) \land (\exists i \in \{f, s\} : x.i \geq x.v \land \text{left}.i = v
\]

\[
\neg \text{is\_oriented}(v, \neg i, i) \quad \rightarrow
\]

\[
\text{right}.v = i
\]

\[
\text{left}.v = \neg i
\]

\[
O_{2} \quad |N_{0}| = 2 \land x.v = X(v) \land \neg \text{is\_decider}(v) \land (\exists i \in \{f, s\} : x.i \geq x.v \land \text{right}.i = v
\]

\[
\neg \text{is\_oriented}(v, i, \neg i) \quad \rightarrow
\]

\[
\text{left}.v = i
\]

\[
\text{right}.v = \neg i
\]
2.2.1 Proof of Correctness for Algorithms $\text{SENS}_1^{\text{chain}}$ and $\text{SENS}_2^{\text{chain}}$

In this section we show that both algorithms $\text{SENS}_1^{\text{chain}}$ and $\text{SENS}_2^{\text{chain}}$ stabilize in at most $n$ rounds to a legitimate configuration (Lemma 2.1, respectively 2.2).

For Algorithm $\text{SENS}_1^{\text{chain}}$, a legitimate configuration satisfies the predicate

$$\mathcal{P}_{\text{SENS}_1} \equiv \{ \forall v, \text{consistent}(v) \land \text{is oriented}(v) \}.$$ 

**Lemma 2.1** After at most $n$ rounds, starting from an arbitrary configuration, Algorithm $\text{SENS}_1^{\text{chain}}$ reaches a configuration where Predicate $\mathcal{P}_{\text{SENS}_1}$ is true.

**Proof.** Each process copies the value of its neighbors and sends it further. Let $\Pi_1 = (v_1, v_2, ..., v_n)$ be a simple path in the chain, and $\Pi_2$ be the reverse of $\Pi_1$, $Pi_2 = (v_n, v_{n-1}, ..., v_2, v_1)$. ($\Pi_1$ and $\Pi_2$ are the only simple paths in the chain of length $n$).

The local state of a process $v_i$ executing Algorithm $\text{SENS}_1^{\text{chain}}$ is the set of values of the write registers $S_{v_i,v_{i-1}}$ and $S_{v_i,v_{i+1}}$. The system configuration is the ordered set of all the local states: $C = ((S_{v_1,v_2}, S_{v_2,v_3}, ..., (S_{v_n,v_{n-1}}))$.

Let $ST_1(C)$ be the set of all the values of registers $S_{v_i,v_{i+1}}$, $1 \leq i < n$, to which we add $ID_{v_1}$ and $ID_{v_2}$.

Let $ST_2(C)$ be the set of all the values of registers $S_{v_{i+1},v_i}$, $1 \leq i < n$, to which we add $ID_{v_1}$ and $ID_{v_2}$.

We define a function $f$ over the system configuration as follows: $f(C) = |ST_1(C)| + |ST_2(C)|$.

Every execution step reduces the value of $f$ by one unit. When $f = 2$, then the system reaches a legitimate configuration:

- if $n > 2$ then $S_{v_1,v_2} = ID_{v_1} \land \forall i \in \{1, ..., n-2\}$, $S_{v_i,v_{i+1}} = S_{v_{i+1},v_{i+2}}$.
- if $n = 2$ then $S_{v_1,v_2} = ID_{v_1}$.

In a legitimate state, all nodes will have left oriented to the node from which $\text{minID}$ has been received and right oriented to the node from which $\text{maxID}$ has been received. □

For Algorithm $\text{SENS}_2^{\text{chain}}$, a legitimate configuration satisfies the predicate

$$\mathcal{P}_{\text{SENS}_2} = \{ \forall \text{nodes } v, x.v = \text{dist}(v) \land \text{is oriented}(v) \}.$$ 

For any node $v$ and any round $t > 0$, let $x[v,t]$ be the value of $x.v$ after the round $t$ is completed.
We show that after $t$ rounds ($t > 0$), $x[v,t]$ is within a certain range (Properties 2.2.1 and 2.2.2). We conclude then within $n/2$ rounds, $x[v,t] = \text{dist}(v)$ (Property 2.2.3).

**Property 2.2.1** For any $t > 0$, within the first $t$ rounds for any node $v$, $x[v,t] \geq X[v]$, where

$$X[v,t] = \min \left\{ \frac{t}{\text{dist}(v)} \right\}$$

**Proof.** Consider the predicate $P(t) :$ within the first $t$ rounds, for any node $v$, $x[v,t] \geq X[v,t]$. We show by induction on $t$ that $P(t)$ holds.

**Basic step** $t = 1$. If node $v$ is one of the extremity nodes then $X[v,1] = 0$. By executing Algorithm $S\hat{E}\hat{N}S_{2\hat{\alpha}}^{\text{chain}}$ after one round $x[v,1] = 0$. If node $v$ is some internal node then $X[v,1] = 1$. By executing Algorithm $S\hat{E}\hat{N}S_{2\hat{\alpha}}^{\text{chain}}$ after one round $x[v,1] \geq 1$. Thus $x[v,1] \geq X[v,1]$, for any node in the chain.

**Inductive step** $t > 1$. By induction hypothesis, after $t - 1$ rounds, $x[v,t - 1] \geq X[v,t - 1]$ and for all $j \in N_v$, $x[j,t - 1] \geq X[j,t - 1]$. We observe that $\min_{j \in N_v} x[j,t - 1] \geq (\min_{j \in N_v} X[j,t - 1])$ thus $(\min_{j \in N_v} x[j,t - 1] + 1) \geq \min_{j \in N_v} (X[j,t - 1] + 1)$. Since

$$\min_{j \in N_v} (X[j,t - 1] + 1) = \min \left\{ \frac{t}{\min_{j \in N_v} \text{dist}(j) + 1} \right\}$$

from the recursive definition of $x[v,t]$ and $\text{dist}(v)$ we obtain that $x[v,t] \geq \text{dist}(v)$. □

**Property 2.2.2** For any $t > 0$, within the first $t$ rounds, for any node $v$, if $\text{dist}(v) < t$, then $x[v,t] = \text{dist}(v)$.

**Proof.** Consider the predicate

$P(t) :$ within the first $t$ rounds, for any node $v$ with $\text{dist}(v) < t$, $x[v,t] = \text{dist}(v)$.

We show by induction on $t$ that the predicate $P(t)$ holds.

**Basic step** $t = 1$. If for some extremity node $v$ the condition $x.v = 0$ does not hold, then in at most one round $x.v$ becomes 0. Thus $P(1)$ holds.

**Inductive step** $t > 1$. Let $v$ be an internal node situated at a distance $\text{dist}(v) < t$. Then there is at least one neighbor $w$ of node $v$ situated at a distance less than $t - 1$ from the extremity nodes of the chain. From the induction hypothesis, $x[w,t - 1] = \text{dist}(w) < t - 1$.

We have two cases, depending on whether the other neighbor $u$ of $v$ is also situated at a distance less than $t - 1$. 

24

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
1) $\text{dist}(u) < t - 1$. Then by induction hypothesis, $x[u, t - 1] = \text{dist}(u)$. From the recursive
definition of $x[v, t]$, we obtain that $x[v, t] = \text{dist}(v)$.

2) $\text{dist}(u) \geq t - 1$. From Property 2.2.1 we know that $x[u, t - 1] \geq X[u, t]$. Since
$\text{dist}(u) \geq t - 1$, we observe that $X[u, t - 1] = t - 1$, thus $x[u, t - 1] \geq t - 1$. Since
$x[w] < t - 1$, from the recursive definition of $x[v, t]$, $x[v, t] = x[w] + 1$, and since
$x[w] = \text{dist}(w)$ thus $x[v, t] = \text{dist}(v)$.

\[\square\]

Property 2.2.3 Within $n/2$ rounds, $x[v, t] = \text{dist}(v)$ and remains so thereafter.

Proof. Every node $v$ in the chain has $\text{dist}(v) \leq n/2$, so by Property 2.2.2, for any round
t > $n/2$, $x[v, t] = \text{dist}(v)$.

\[\square\]

Lemma 2.2 After at most $n$ rounds, starting from an arbitrary configuration, Algorithm
$SENS_2^{\text{chain}}$ reaches a configuration where Predicate $P_{SENS_2}$ is true.

Proof. By Property 2.2.3, within $n/2$ rounds, $x[v, t] = \text{dist}(v)$ and remains so as long as
no topology change occurs (Guard $X$ remains disabled for the rest of the execution).

Once the $x$ values stabilize, there exists a unique decider node $dn$ (for which the predicate
is decider ($dn$) is evaluated to true), and for any other node $v$ there exists a unique neighbor
$nb$ that is closer to node $nd$.

After at most one more round, node $dn$ decides the left and right orientation for itself,
by executing either Guard $D_e$ or $D_o$, depending on whether the chain has even, respectively
odd number of nodes. Since $x$ does not change anymore, and predicate has orientation is
evaluated to true, Guard $D_e$ or $D_o$ is not enabled anymore for the rest of the execution, as
long as no topology change occurs. So the node $dn$ becomes disabled for any future round.

Each node $v$ then decides its orientation based on the node $nb$, by executing both guards
$O_l$ and $O_r$. Once executed, the guards $O_l$ and $O_r$ are not anymore for the rest of the
execution, as long as no topology change occurs. So the node $v$ becomes disabled for any
future round.

Thus, within $n/2$ rounds, for all $v \in V$, the following predicate is true: $(\text{is decider}(v) \land
\text{has orientation}(v)) \lor (\neg \text{is decider}(v) \land \text{is oriented}(v, v_{\text{left}}, v_{\text{right}}))$.

\[\square\]
2.3 Algorithms for Orienting an Unoriented Ring

In this section we propose two algorithms for orienting an unoriented ring with a distinguish node (called leader) and with at least three nodes \((n \geq 3)\). Their proof of correctness is similar to the proof of correctness of algorithms \(\mathcal{ENS}_1^{\text{chain}}\) and \(\mathcal{ENS}_2^{\text{chain}}\) in Section 2.2.1.

Algorithm \(\mathcal{ENS}_1^{\text{ring}}\) is implemented in the link-register model. Algorithm \(\mathcal{ENS}_2^{\text{ring}}\), implemented in the shared-memory model, also requires that the nodes 'IDs are at least chromatic \([KY02]\).

Algorithm \(\mathcal{ENS}_1^{\text{ring}}\) (Figure 2.3) works as follows. The leader decides the orientation of the ring as follows. It considers its first neighbor \(f\) as its left and its second neighbor \(s\) as its right, and it sends two values: value 0 to node \(f\) and value 1 to node \(s\). These values are forwarded by the other nodes. A node consider its right as the node from which the smallest value has been received, and as its left the node from which the largest value has been received (see Figure 2.3).

A node \(u\) will write into registers \(R_f, R_s\) what it is has to communicate to the first, respectively second neighbor. Also, node \(v\) will read from registers \(R'_f, R'_s\) what the first, respectively second neighbor has sent to it. By abuse of notation, for the two neighbors \(f\) and \(s\) of some node \(v\), \(-f = s\) and \(-s = f\).

![Figure 2.3: Ring orientation in the link-register model](image)

Predicate \(\text{consistent}(v)\) is true when node \(v\) has its registers consistent with its neighbors' registers.
Algorithm 2.3.1 Self-Stabilizing Orientation of a Ring $\mathcal{S}\mathcal{E}\mathcal{N}\mathcal{S}_1^{\text{ring}}$

**Predicate** \(\text{consistent}(v) \equiv (v = L \land R_f = 0 \land R_s = 1) \lor (v \neq L \land \forall i \in \{f,s\} R_i = R'_{i-1})\)

**Actions executed by the ring leader** \(L\)

\[
\begin{align*}
 b & \quad (R_f \neq 0 \lor R_s \neq 1) \rightarrow \\
 & \quad \text{if } (R_f \neq 0) \text{ then } R_f = 0 \\
 & \quad \text{if } R_f \neq 0 \text{ then } R_s = 1
\end{align*}
\]

\(O\)

\[
\begin{align*}
 l_L &= f \\
 r_L &= s
\end{align*}
\]

**Actions executed by any other node of the chain** \(v \neq L\)

\[
\begin{align*}
 b & \quad \exists i \in \{f,s\} : R_i \neq R'_{i-1} \rightarrow R_i = R'_{i-1}
\end{align*}
\]

\(O\)

\[
\begin{align*}
 \text{consistent}(v) \land \exists i \in \{f,s\} R'_i > R'_{i-1} & \rightarrow \\
 l_v &= i \\
 r_v &= -i
\end{align*}
\]

Algorithm $\mathcal{S}\mathcal{E}\mathcal{N}\mathcal{S}_1^{\text{ring}}$ is uniform and does not terminate (Guard \(O\) is executed infinitely often). The space complexity in a node is independent of \(n\) (depends only on the size of nodes' IDs). The stabilization time is at most \(n\).

Algorithm $\mathcal{S}\mathcal{E}\mathcal{N}\mathcal{S}_2^{\text{ring}}$ works as follows. Every node in the ring other than the ring leader has a certain distance to the ring leader. For the ring leader, \(dist(L) = 0\). For any other node \(v\), \(dist(v) = \min_{j \in N_v} dist(j) + 1\) (see Figure 2.4).

![Figure 2.4: dist values for a ring with 9 nodes](image)

27

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The ring leader decides the orientation of the ring (it is a so called decider). The decider has a double purpose. One purpose is to be the only one to start the decision process on the edges' orientation. The second purpose is to help a node in selecting the neighbor to follow in deciding the orientation of the edges adjacent to it (every other node will select the neighbor closer to the decider).

Every node holds a variable $x.v$. The set of all the $x$ values for the entire ring can be considered as an array $x$ (instead of $x.v$ we can write $x[v]$), the first element being $x.L$. Let $X(v)$ be an abstract function of the physical values $x$ of the neighbors of node $v$ defined as follows:

(i) if $v = L$ (if $v$ is the ring leader), then $X(v) = 0$.

(ii) if $v \neq L$, then

$$X(v) = \max \left\{ 1, \min_{j \in N_v} x[j] + 1 \right\}$$

Starting from an arbitrary state, each node in the ring sets its $x$ value to $X(v)$ and eventually converges to $dist(v)$ in finite time.

Algorithm $SENS_2^{ring}$ is uniform, deterministic, and is presented next (Figure 2.3).

Guard $X$ is responsible for making sure that $x.v = X(v)$. When the ring leader has decided a correct orientation for itself (which is its left and which is its right neighbor), Predicate $has\_orientation(L)$ is evaluated to true. When a node other than the ring leader has a correct orientation, Predicate $is\_oriented(v, l_v, r_v)$ is evaluated to true. Guards $O_l$ and $O_r$ are executed by a non-leader node when it has no correct orientation. Then it decides the orientation based on the orientation of one of its neighbors that is closer to the leader node (the $x$ value of that neighbor is less or equal to the node's $x$ value).
Algorithm 2.3.2 Self-Stabilizing Orientation of a Ring $SENS^r_{ring}$

Predicates
\[ is\_oriented(l, v, v_{left}, v_{right}) \equiv l = v_{left} \land r = v_{right} \]
\[ has\_orientation(v) \equiv l_v = f \land r_v = s \]

Actions executed by the ring leader $L$
\[
X, O \quad \neg(x.L = 0 \land has\_orientation(l_L, r_L)) \quad \rightarrow \\
\begin{align*}
x.L &= 0 \\
l_L &= f \\
r_L &= s
\end{align*}
\]

Actions executed by any other node in the ring $v \neq L$
\[
X \quad x.v \neq X(v) \quad \rightarrow \\
\begin{align*}
x.v &= X(v) \\
l_v &= r_v = \bot
\end{align*}
\]
\[
O_l \quad x.v = X(v) \land (\exists i \in \{f, s\} : x.i \leq x.v \land left.i = v \land \neg is\_oriented(l_v, r_v, \neg i, i)) \quad \rightarrow \\
\begin{align*}
r_v &= i \\
l_v &= \neg i
\end{align*}
\]
\[
O_r \quad x.v = X(v) \land (\exists i \in \{f, s\} : x.i \leq x.v \land right.i = v \land \neg is\_oriented(l_v, r_v, i, \neg i)) \quad \rightarrow \\
\begin{align*}
l_v &= i \\
r_v &= \neg i
\end{align*}
\]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 3

SIMULTANEOUSLY ACTIVATED PROCESSES ON A CHAIN

In this chapter we present Algorithm $SSDS$ (Section 3.1), followed by its proof of correctness (Section 3.1.1), and then give one application of the proposed algorithm: Algorithm $LMEC$ (Section 3.2). The other application of Algorithm $SSDS$, Algorithm $SORT_c$, is presented in Chapter 4. In Section 3.3 we show why Algorithm $SSDS$ cannot be simply extended to the read/write atomicity model mentioned in Chapter 1. Algorithm $SSDS$ is a general self-stabilization scheme for simultaneously activated non-adjacent processes on an asynchronous oriented chain. It is uniform and works under the unfair distributed daemon. It uses one bit per node, and stabilizes in at most $n - 1 + 2t$ rounds to the global predicate, i.e., every node has executed at least $t$ times, $t \geq 0$. For a synchronous system, after $n - 1$ steps, every node is enabled every second round. If the synchronous network starts in a normal starting configuration, then a node is active every other round from the beginning. We then give two applications of the proposed algorithm: a space and time optimal solution to the local mutual exclusion problem that satisfies the strong safety property – in any configuration, there exists at least one privileged node (Algorithm $LMEC$), and a space and asymptotically time optimal solution to the distributed sorting problem on the oriented chain where the values to be sorted are not necessarily distinct, and each process holds only one value at any moment (Chapter 4 - Algorithm $SORT_c$).

3.1 Self-Stabilizing Distributed Simultaneous Execution of Non-adjacent Nodes on an Oriented Chain $SSDS$

Algorithm $SSDS$ is illustrated in Figure 3.1.

Each node holds a variable $S \in \{A, B\}$, and thus needs only one bit. A node is enabled
to execute if the following two conditions are true: (i) either it has no left neighbor, or the left neighbor $S$ value is different to its $S$ value, and (ii) either it has no right neighbor, or the right neighbor $S$ value is the same as its $S$ value.

For any node $v$, let $l_v, r_v$ be the left, respectively right neighbor of node $v$, $S = S_v, S_l = S.l_v, S_r = S.r_v$. Predicate $\text{check}(v)$ has as parameter a node ID and is responsible for checking whether the given node $v$ exists, and if so, whether it has a certain value for its state (variable $S$). Macro $\text{execute}(v)$ is a generic macro: node $v$ executes something based on its values and/or the values of its neighbors.

**Algorithm 3.1.1 Algorithm SSDS**

**Predicate** $\text{check}(v, s) \equiv (v = 1 \lor S.v = s)$

**Actions for any node $v$**

$$
\begin{align*}
\text{ABB} & \quad S = B \land \text{check}(l_v, A) \land \text{check}(r_v, B) & \rightarrow & \text{execute}(v) ; \quad S = A \\
\text{BAA} & \quad S = A \land \text{check}(l_v, B) \land \text{check}(r_v, A) & \rightarrow & \text{execute}(v) ; \quad S = B
\end{align*}
$$

Consider the network of seven nodes in Figure 3.1.

![Network Diagram]

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

Figure 3.1: Four steps in a synchronous system

In the starting configuration shown in Figure 3.1(a), the only enabled nodes are the
odd numbered nodes. If we assume a synchronous system, the next execution step brings
the system into the configuration in Figure 3.1(b), in which the enabled nodes are the even
numbered nodes. After one more synchronous round, the system reaches the configuration
shown in Figure 3.1(c), the same as the initial configuration. This cycle repeats forever.

3.1.1 Proof of Correctness for Algorithm SSDS

In this section, we prove that Algorithm SSDS stabilizes in at most $n - 1 + 2k$ rounds
to the global predicate

$k$-Execute $\equiv \{ \forall \text{node } v, v$ has executed macro execute at least $k$ times $\}$

and also works under the unfair distributed daemon (Property 3.1.8, Section 3.1.1.1).

We define the notions of configuration-string and difference-string, and prove some prop­
erties of Algorithm SSDS. We then show that by executing Algorithm SSDS under the
weakly fair distributed daemon, the following properties are true:

- Only one node per neighborhood is enabled (local mutual exclusion) (Property 3.1.2)
- At least one node is enabled (no deadlock) (Property 3.1.3)
- After it executes, it remains disabled until all its neighbors execute (1-fairness) (Property
  3.1.4)
- In at most $n - 1 + 2t$ rounds, every node has executed at least $t$ times (no starvation)
  (Lemma 3.1).

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

Definition 3.1.1 (Configuration-String) We call a configuration-string the string ob­
tained by concatenating the states of all the nodes (variable $S$), from left to right.

A normal starting configuration refers to a configuration in which the corresponding
configuration string is $(AABB)^n$, i.e., a string of length $4n$ obtained by concatenating $n$
copies of $AABB$. Note that in a normal starting configuration, the odd numbered nodes
are enabled (see Figure 3.1(a)). In a synchronous system, a normal starting configuration is
reachable from any configuration in at most $n - 1$ steps. Since there is a bijection between
configurations and configuration-strings, We sometimes simply say “configuration” to mean
“configuration-string.”

32
Any configuration of length \( n > 1 \) can be mapped to an unique binary \((n-1)\)-bit string, called \textit{difference-string}, defined below.

**Definition 3.1.2** Given \( C = S_1S_2\ldots S_n \) an \( n \)-length configuration, we call a difference-string the \((n-1)\)-length binary string \( D_S = b_1b_2\ldots b_{n-1} \) such that \( b_i = 0 \) if \( S_i = S_{i+1} \), 1 otherwise.

For example, the difference-string of \( ABBAABBAAA \) or \( BABBABBBB \) is 101010100.

**Remark 3.1** Given a difference-string \( D_S \) and the value \( S \) of some node, the corresponding configuration \( C \) is uniquely defined.

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

From the code of Algorithm SSDS, we observe that:

**Observation 3.1.1** Given any configuration \( C \):

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

(ii) Guard \( ABB \) or \( BAA \) is enabled at the rightmost node \( v \), if and only if \( D_S(v) \) is 1 and the execution of the guard changes it from 1 \( \rightarrow \) 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 \( D_S(v) \) is 10 and the execution of the guard changes it from 10 \( \rightarrow \) 01.

**Property 3.1.2** 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 3.1.1(i), \( D_S(v) = 0 \), thus \( D_S(r_v) \) starts with a 0. From Observation 3.1.1(ii, iii), for node \( r_v \) to be enabled to execute, \( D_S(r_v) \in \{10, 1\} \), thus it should start with a 1. Contradiction.
2) $v$ is the rightmost node $R$. Then, from Observation 3.1.1(ii), $DSC(v) = 1$, thus $DSC(l_v)$ ends with a 1. From Observation 3.1.1(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.

3) $v$ is a node other than $L$ or $R$. Then, from Observation 3.1.1(iii), $DSC(v) = 10$, thus $DSC(l_v)$ ends with a 1 and $DSC(r_v)$ starts with a 0. From Observation 3.1.1(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 3.1.1(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. □

**Property 3.1.3** 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 3.1.1(i) the leftmost node $L$ is enabled to execute.

2) If $DSC$ starts with a 1 and contains the substring 10, then by Observation 3.1.1(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 3.1.1(ii), the rightmost node $R$ is enabled to execute. □

**Property 3.1.4** 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 3.1.1(i,ii), if node $v$ is either the leftmost or the rightmost node, after executing its enabled guard, it becomes disabled. From Observation 3.1.1(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. □

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

Given two nodes $v$ and $r_v$ with $S.v = a$ and $S.r_v = b$, by notation "$a \rightarrow b$" 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 defining the layers of nodes from $L$ from 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 starting from the node $L$ and going to the rightmost node. The peak nodes are the enabled nodes.

The difference-string of a given configuration is consistent with the orientation of the arrow between consecutive $S$ values (1 for $\rightarrow$, 0 for $\leftarrow$).

For example, for the configuration $BAAAABBABBBABAA$ of a 16-node network, the level ordering of the nodes is illustrated in Figure 3.2(a).

Definition 3.1.3 (Node Delay) 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 for a node is the maximum number of rounds a node must to wait until it becomes enabled. The delay values of the nodes in Figure 3.2(a) are given in Figure 3.2(b).
After a node executes, the arrows (or the arrow, for the chain extremities) to that node are reversed. The delay values are then recalculated.

**Property 3.1.5** In any configuration, if $w$ is a neighbor of $v$ then $\text{delay}[w] = \text{delay}[v] \pm 1$.

**Proof.** From Definition 3.1.3. □

**Property 3.1.6** 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 $(t+1)^{st}$ 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 3.1.4, 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 $d_0$. By the definition of array delay, $1 \leq D_0 \leq n - 1$.

**Lemma 3.1** 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 $q \geq 1$ that Predicate $P(q)$ 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 Property 3.1.5, $d_0[u] = d_0[v] \pm 1$. 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 Property 3.1.6, 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 Property 3.1.6, 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. \hfill \Box

3.1.1.1 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 to execute but not selected by the distributed daemon, it remains enabled (Property 3.1.7). Since the unfair daemon has to select a non-empty subset of the enabled nodes in every computation step, it will be forced to select $v$ (Property 3.1.8).
**Property 3.1.7** If a node $v$ is enabled to execute but is not selected by the daemon, it remains enabled until it is selected.

**Proof.** If some node $v$ is enabled, by Property 3.1.2, neither of the existing neighbors is enabled. Since $v$ is not selected by the daemon to execute, the neighboring nodes remain disabled until $v$ is selected. □

**Property 3.1.8** Every continuously enabled node will be eventually selected by the unfair distributed daemon after finite number of rounds.

**Proof.** By contradiction. Assume that there exists a node $v$ in the chain that is continuously enabled but the unfair daemon never selects it for execution. Since the executions of Algorithm SSVS are 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, and $v \notin A$.

If node $u$ executes infinitely often, then both neighbors of $u$ execute infinitely many often (Property 3.1.4, Lemma 3.1). Thus, if $u \in A$, then $left(u), right(u) \in A$. By induction, $A$ consists of all nodes. Contradiction. □

### 3.2 Self-Stabilizing Local Mutual Exclusion Algorithm on Oriented Chains LMEC

Each node holds two variables: variable $S$ that takes values in the set $\{A, B\}$, and Boolean variable $request$ that is true whenever the process requests access to its critical section $CS$. For some node $v$, let $S = S.v$ and $request = request.v$. Predicate $\tilde{v}$ has been defined in Section 3.1.

#### 3.2.1 Proof of Correctness of Algorithm LMEC

A protocol solves the local mutual exclusion problem if any configuration of the system running the protocol has two properties ([AN02]): (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.
Algorithm 3.2.1 Algorithm $\mathcal{LMEC}$

**Actions for any node** $v$

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

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

Property 3.1.2 shows that Algorithm $\mathcal{LMEC}$ has the safety property. Lemma 3.1 shows that Algorithm $\mathcal{LMEC}$ has the liveness property.

3.3 Algorithm $\mathcal{SSDS}$ 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: $bAa$.

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:

  $\begin{align*}
  Aa & \rightarrow Ba \\
  Bb & \rightarrow Ab \\
  bA & \rightarrow bB \\
  aB & \rightarrow aA \\
  bAa & \rightarrow bBa \\
  aBb & \rightarrow aAb \\
  *a & \rightarrow *b \\
  *b & \rightarrow *a \\
  a* & \rightarrow b* \\
  b* & \rightarrow a* \\
  **a & \rightarrow **b \\
  **b & \rightarrow **a \\
  a** & \rightarrow b** \\
  b** & \rightarrow a**
  \end{align*}$

  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:
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:

- $* * * AabB * * * \rightarrow * * * AbaB * * *$
- $* * bAa * Ab * A \rightarrow * * bBa * Aa * A$
Changes that might seem to be allowed but are not:

\[ A \star b \star a \star B \rightarrow A \star a \star b \star B \]

although each can be accomplished in two steps:

\[ A \star b \star a \star B \rightarrow A \star a \star a \star B \rightarrow A \star a \star b \star B \]

The edge codes can be divided into good and bad edges.

There are 16 possible edge codes; 8 are good and 8 are bad:

- AaaA good
- AaaB bad
- AabA bad
- AabB bad
- AbaA good
- AbaB good
- AbbA bad
- AbbB good
- BaaA good
- BaaB bad
- BaaB 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 SSSS 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 critical section (every edge is bad). Consider the following possible path of execution of some in the read/write atomicity protocol based on the Algorithm SSSS 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.

\[
\text{AaaBbaBbbAabA} \rightarrow \\
\text{BaaBbaAbbAabB} \rightarrow
\]
The execution ends in the starting configuration, without reaching a good (legitimate) state. We conclude that the convergence does not hold for this model.
CHAPTER 4

OPTIMAL SORTING ON A CHAIN

In this chapter we present an application of Algorithm SSDS: the distributed sorting problem on the oriented chain where the values to be sorted are not necessarily distinct, and each process holds only one value at any moment.

We present two solutions: a sorting algorithm for an abstract model of communication (Algorithm \(A_{\text{SORT}}_c\)), then we show how sorting will be done in the shared memory model of communication (Algorithm \(\text{SORT}_c\)).

Using some properties of Algorithm SSDS, we show that Algorithm \(A_{\text{SORT}}_c\) stabilizes in at most \(2n - 2\) rounds under the unfair distributed daemon (Section 4.2). We then show that Algorithm \(\text{SORT}_c\) reduces to Algorithm \(A_{\text{SORT}}_c\) (Section 4.3), thus it sorts \(n\) values on a chain in non-decreasing order from left to right, in at most \(4(2n - 2)\) rounds. It uses a total of three bits per node, and is thus an improvement over the algorithms of [Sas02, Sas04].

4.1 Self-Stabilizing Distributed Sorting Algorithms in an Oriented Chain

In this section we present two algorithms for distributed sorting problem in an oriented chain: \(A_{\text{SORT}}_c\) (Section 4.1.1), and \(\text{SORT}_c\) (Section 4.1.2). Algorithm \(A_{\text{SORT}}_c\) is implemented in an abstract model. Algorithm \(\text{SORT}_c\) is implemented in the shared memory model.

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

1. \(x = x + y\)
2. \(y = x - y\)
3. \(x = x - y\)

If \(x\) and \(y\) are two bit-strings, then the swapping can be done bit-wise as follows:

1. \(x = x \oplus y\)
2. \(y = x \oplus y\)
3. \(x = x \oplus y\)
4.1.1 Distributed Sorting in an Oriented Chain

Each node, besides the variable $S$, holds one variable $IV$ to be sorted. Algorithm $ASORT_c$ (Figure 4.1.1) 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 $IV.r_v$ in certain situations. Intuitively, since by executing Algorithm $SSDS$, local mutual exclusion is valid in any configuration (see Property 3.1.2), a node can synchronize the swap with its right neighbor, and once done, pass the token to it. We assume for now that the swap is done in an atomic step (macro $swap$), and we show in Section 4.1.2 how this is done in the shared memory model.

For some node $v$, let $S = S.v$, $S_l = S.l_v$, $S_r = S.r_v$. Predicate $check(v)$ has been defined in Section 3.1.

\begin{algorithm}
\textbf{Algorithm 4.1.1} Self-Stabilizing Distributed Sorting in an Oriented Chain in the Abstract Model $ASORT_c$

\begin{align*}
\text{Macro } swap(v, w) &:: \text{ if } (w \neq 1 \land IV.v > IV.w) \text{ then } \\
& \quad IV.v = IV.v + IV.w; \ IV.w = IV.v - IV.w; \ IV.x = IV.v - IV.w \\
\text{Sorting actions for any node } v \\
AB & S = B \land check(l_v, A) \land check(r_v, B) \rightarrow swap(v, r_v); \ S = A \\
BA & S = A \land check(l_v, B) \land check(r_v, A) \rightarrow swap(v, r_v); \ S = B
\end{align*}

The sorting actions are mutually exclusive, so Algorithm $ASORT_c$ is deterministic.

4.1.2 Sorting in the Shared-Memory Model

In Algorithm $SORT_c$ (Figure 4.1.2), each node $v$ holds three variables: variable $IV$ to be sorted, variable $S \in \{A, B, X, Y\}$, and variable $tmpS \in \{A, B\}$. Variable $tmpS$ stores the value of variable $S$ temporarily while the swap is performed.

For some node $v$, let $S = S.v$, $IV = IV.v$, $tmpS = tmpS.v$, $S_l = S.l_v$, $IV_l = IV.l_v$,
Macro $\text{swap}'(v, r_v, value)$ 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 $\text{tmp}S.v$. Predicate $\text{check}(v)$ has been defined in Section 3.1.

**Algorithm 4.1.2 Self-Stabilizing Distributed Sorting in an Oriented Chain in Shared Memory Model $\text{SORT}_c$**

Macro $\text{swap}'(v, w, tS) : :$ if $(w \neq \bot \land IV.v > IV.w)$ then 
\[ \text{tmp}S.v = tS; \quad IV.v = IV.v + IV.w; \quad S.v = X \]

**Sorting actions for any node $v$**

- **ABB** \[ S = B \land \text{check}(l_v, A) \land \text{check}(r_v, B) \rightarrow \text{swap}'(v, r_v, A) \]
- **BAA** \[ S = A \land \text{check}(l_v, B) \land \text{check}(r_v, A) \rightarrow \text{swap}'(v, r_v, B) \]

**Synchronizing actions for any node $v$**

- **S1** \[ S \in \{A, B\} \land l_v \neq \bot \land S_l = X \rightarrow IV = IV_l - IV; \quad \text{tmp}S = S; \quad S = Y \]
- **S2** \[ S = X \land r_v \neq \bot \land S_r = Y \rightarrow IV = IV - IV_r; \quad S = \text{tmp}S \]
- **S3** \[ S = Y \land l_v \neq \bot \land S_l \neq X \rightarrow S = \text{tmp}S \]
- **C1** \[ S = Y \land l_v = \bot \rightarrow S = \text{tmp}S \]
- **C2** \[ S = X \land r_v = \bot \rightarrow S = \text{tmp}S \]
- **C3** \[ S = X \land r_v \neq \bot \land S_r = X \rightarrow S = \text{tmp}S \]

In order to perform the swap, nodes $v$ and $r_v$ need to change their values of variable $S$ from either $A$ or $B$ to either $X$ or $Y$. Since node $v$ will change the value of its $S$ after the swap, the value to-be for $S.v$ and the value of $S.r_v$ are stored in variables $\text{tmp}S.v$, respectively $\text{tmp}S.r_v$ by each node. Node $v$ changes its $S$ to $X$ (macro $\text{swap}'$) and node $r_v$ changes its $S$ to $Y$ (Guard $S1$). The swap started by node $v$ already in macro $\text{swap}'$ is continued by node $r_v$ in Guard $S1$, and finished by node $v$ in Guard $S2$ (where also it restores its $S$). Once the swap is done, their $S$ values are restored back to $A$ or $B$, node $v$
in Guard $S_2$, node $r_v$ in Guard $S_3$.

In Figure 4.1, nodes $v$ and $r_v$ need to swap their values. The state of each node is in order $S; IV; tmpS$.

![Diagram](image)

Figure 4.1: Nodes $v$ and $r_v$ swap their $IV$ values

The synchronizing actions $S_1-C_3$ are mutually exclusive with the sorting actions, and mutually exclusive among themselves, so Algorithm $SORT_c$ is deterministic also.

4.2 Proof of Correctness of Algorithm $A.SORT_c$

Besides local mutual exclusion, sorting requires synchronization between neighboring nodes. Each node has a so-called pseudo-time such that the comparison (and eventual swapping) between two neighboring nodes is done when the nodes have the same pseudo-time values.

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

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 $C_0$. Function $\Psi_0$ is defined as follows: (i) the leftmost node $L$ has the same $\Psi$ value as its right neighbor, $\Psi_0(1) = \Psi_0(2)$, and (ii) given two neighboring nodes $v$ and its left neighbor $l_v$ with positions $i$ and $i - 1$, $\Psi_0(i) = \frac{d_0(v) + d_0(l_v) - 1}{2}$.

For example, given the configuration in Figure 3.2(a), the $\Psi_0$ values are given in Figure 4.2(a)).
If the node at position \( i \) is enabled, then \( \Psi_0(i) = \Psi_0(i + 1) \) (if \( i \leq n - 1 \)).

**Definition 4.2.1** 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 node at position \( i \) has executed during this step then \( \Psi_j(i) \) and \( \Psi_j(i + 1) \) increase by 1:

\[
\Psi_{j+1}(i) = \Psi_j(i) + 1 \quad \text{and} \quad \Psi_{j+1}(i + 1) = \Psi_j(i + 1) + 1.
\]

Additionally, if node \( r_L \) executes, then node \( L \) increases its pseudo-time also \( \Psi_{j+1}(1) = \Psi_j(1) + 1 \).

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

For example, given \( \Psi_0 \) from Figure 4.2(a), if the marked nodes execute, then the next pseudo-time values are the ones in Figure 4.2(b).

**Observation 4.2.1** The following relations hold:

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

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

\[
\Psi_0(i) \leq \max \left\{ \frac{n-i}{i-2} \right\}
\]

**Corollary 4.2.2** \( \Psi_0(i) \leq n - 2 \), 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 4.2.3** \text{parity} is a global constant.

**Property 4.2.4** \( \mathcal{E}(i,t) \) holds if and only if \( t \geq \Psi_0(i) \) and \( i + t + \text{parity} \) is even.
Proof. From Definition 4.2.1 we observe that:

- If $E(i, t)$ is true then $E(i, t + 2k + 1)$ is false and $E(i, t + 2k)$ is true, for all $k \geq 0$.
- If $E(i, t)$ is false then $E(i, t + 2k + 1)$ is true and $E(i, t + 2k)$ is false, for all $k \geq 0$. □

Given a starting configuration $C_0$ and $C_j$ some configuration after Algorithm $A_{\text{SORT}}_c$ has executed a number of steps, the relationship between the number of rounds that have elapsed and $\Psi_j$ is given by Property 4.2.5.

Property 4.2.5 Given a starting configuration $C_0$ and $C_j$ some configuration after Algorithm $A_{\text{SORT}}_c$ has executed a number of steps, then the number of rounds elapsed is $q \leq \min\{1 \leq i \leq n, \Psi_j(i)\}$.

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 rank [CLRS01]. The 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 pos with two parameters as follows.

Definition 4.2.2 Given $r$, $1 \leq r \leq n$, and value $t \geq 0$, the value $\text{pos}[r,t]$ represents the position of the value of rank $r$ when the node $v$ at position $i$ that holds the value has the current $\Psi(i) = t$.

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

Lemma 4.1 proves that within $2n - 2$ pseudo-steps, the values in the chain are sorted.

Lemma 4.1 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 $r \in 1 \ldots n$, where $\text{pos}[r, t]$ represents the position of the value of rank $r$ at pseudo-time $t \geq 0$. 

49

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Proof. (a) For any position $i \in 1 \ldots n$, for any $t \geq \Psi_0(i)$, Predicate $E(i, t)$ is true if and only if $i + t + \text{parity}$ is even.

Since $\forall i \in 1 \ldots n$, $\Psi_0(i) \leq n - 2$, it results that $\forall t \geq n - 2$, Predicate $E(i, t)$ 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 $P(n)$ 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 $P(n)$ holds.

Let $S.i.t$ and $IV.i.t$ be the value of variable $S$, respectively $IV$, of the node at position $i$ in the $n$-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 $P(n - 1)$ is true and we show that Predicate $P(n)$ is true. We assume that at time $t = 0$ all alternated nodes are enabled and let $C$ be such a configuration.

Property 4.2.6 shows that the maximum value moves to the last position.

**Property 4.2.6**

$$\text{For any } t \geq 2, \text{pos}[n, t] = \min \left\{ \begin{array}{ll} \text{pos}[n, t - 1] + 1, & \text{if } \text{pos}[n, t - 1] < n \\ n, & \text{if } \text{pos}[n, t - 1] = n \end{array} \right.$$  

**Proof.** Let $p$ be the position of the maximum element at pseudo-time 0: $\text{pos}[n, 0] = p$. If $p = n$ then $\forall t \geq 0, \text{pos}[n, t] = n$. Assume $p < n$. Predicate $E(p, 0)$ can be either true or false.

- Predicate $E(p, 0)$ 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: $\text{pos}[n, 1] = \text{pos}[n, 0] + 1 = p + 1$. Then $\forall t \geq 2$ such that $p + t \leq n$, Predicate $E(p + t, t)$ is true, and $\text{pos}[n, t] = \text{pos}[n, t - 1] + 1$, if $\text{pos}[n, t - 1] < n$.

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

50

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Corollary 4.2.7 $pos[n, n] = n$.

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 $E'(i, t)$ is: "The node at position $i$ in the $(n - 1)$-node chain is enabled at the pseudo-time $t$." We show in Property 4.2.8 that Predicate $E'(i, 0)$ depends on $E(i, 1)$ and $E(i + 1, 0)$.

**Property 4.2.8**

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

**Proof.** It follows from definition of $S'.i.0$. $\square$

**Corollary 4.2.9** 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 4.2.10** $\text{parity}' = 1 - \text{parity}$. 

51
Proof. For \( i < \text{pos}[n,1] \), 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 \( i + 1 > \text{pos}[n,0] \), 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 \( \forall i \in 1...n-1 \), Predicate \( \mathcal{E}'(i,0) \) is true if and only if \( i + 1 + \text{PARITY} \) is even, or \( i + 1 - \text{PARITY} \) is even. \( \square \)

Property 4.2.11 For all \( t \geq 0 \),

\[
\mathcal{E}'(i,t) \equiv \begin{cases} 
\mathcal{E}(i,t+1), & \text{if } i < \text{pos}[n,t+1] \\
\mathcal{E}(i+1,t), & \text{if } i + 1 > \text{pos}[n,t] 
\end{cases}
\]

\[
S'.i.t \equiv \begin{cases} 
S.i.(t+1), & \text{if } i < \text{pos}[n,t+1] \\
\text{reverse}(S.(i+1).t), & \text{if } i + 1 > \text{pos}[n,t] 
\end{cases}
\]

\[
 IV'.i.t \equiv \begin{cases} 
IV.(i+1).t, & \text{if } i + 1 > \text{pos}[n,t] \\
 IV.(i.t), & \text{if } i < \text{pos}[n,t+1] 
\end{cases}
\]

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 4.2.8.

Inductive step \( t > 0 \).

Predicate \( \mathcal{E}'(i,t) \) 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 \( \mathcal{E}'(i,t) \) is equivalent to Predicate \( \mathcal{E}(i,t+1) \) when \( i < \text{pos}[n.t+1] \), and also equivalent to Predicate \( \mathcal{E}(i+1,t) \) when \( i + 1 > \text{pos}[n,t] \).

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:

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

and

\[
 IV'.i.t \equiv \begin{cases} 
 IV'.i.(t-1), & \text{if } i \text{ is even and } IV'.i.(t+1).(t-1) > IV'.i.(t-1) \\
 IV'.(i+1).(t-1), & \text{if } i \text{ is even and } IV'.i.(t+1).(t-1) < IV'.i.(t-1) \\
 IV'.i.(t-1), & \text{if } i \text{ is odd and } IV'.i.(t-1) > IV'.i.(i-1).(t-1) \\
 IV'.(i-1).(t-1), & \text{if } i \text{ is odd and } IV'.i.(t-1) < IV'.i.(i-1).(t-1) 
\end{cases}
\]
At pseudo-time $t-1$, all odd-position nodes in the $n$-node chain are enabled, thus:

$$S_i(t) = \begin{cases} S_i(t-1), & \text{if } i \text{ is even} \\ \text{reverse}(S_i(t-1)), & \text{if } i \text{ is odd} \end{cases}$$

and

$$IV_i(t) = \begin{cases} IV_i(t-1), & \text{if } i \text{ is odd and } IV_i(t-1) \leq IV_i(t-1) \\ IV_{i+1}(t-1), & \text{if } i \text{ is odd and } IV_i(t-1) > IV_{i+1}(t-1) \\ IV_{i-1}(t-1), & \text{if } i \text{ is even and } IV_i(t-1) \geq IV_{i-1}(t-1) \\ IV_i(t-1), & \text{if } i \text{ is even and } IV_i(t-1) < IV_{i-1}(t-1) \end{cases}$$

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

$$S_i(t+1) = \begin{cases} S_i(t), & \text{if } i \text{ is odd} \\ \text{reverse}(S_i(t)), & \text{if } i \text{ is even} \end{cases}$$

and

$$IV_i(t+1) = \begin{cases} IV_i(t), & \text{if } i \text{ is even and } IV_i(t+1) \geq IV_i(t) \\ IV_{i+1}(t), & \text{if } i \text{ is even and } IV_i(t+1) < IV_{i+1}(t) \\ IV_i(t), & \text{if } i \text{ is odd and } IV_i(t) \geq IV_{i+1}(t) \\ IV_{i-1}(t), & \text{if } i \text{ is odd and } IV_i(t) < IV_{i-1}(t) \end{cases}$$

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: $IV'_i.t \leq IV'_i.i+1.t, \forall i \in 1, \ldots, n-1$. Since $pos[n,n-1] \geq n-1$, by Property 4.2.11, $IV'_i.n-1 = IV_i.n$, where $t = n-1$. Thus the values in the $n$-node chain are sorted. □

4.3 Reduction of Algorithm $ASORT_c$ in Abstract Model to Algorithm $SORT_c$ in Shared-Memory Model

In this section we first show that Algorithm $SORT_c$ reduces to Algorithm $ASORT_c$. We can then conclude that, starting from an arbitrary configuration, in at most $10n - 12$ rounds, Algorithm $SORT_c$ sorts the values in non-decreasing order (Lemma 4.4).

We use Definition 2.1.1 of reduction from Chapter 2.

If $A_2$ accomplishes a task in the model $M_1$ and $A_1$ reduces to $A_2$, then by Definition 2.1.1, $A_1$ accomplishes the same task in the model $M_2$.

We now show that Algorithm $SORT_c$ reduces to Algorithm $ASORT_c$. 53
Let $S_v = (s_v, x_v)$ be the set of all variables of node $v$ in order $(S, IV)$ used by Algorithm $\text{ASORT}_c$ in the abstract model. Let $S'_{v} = (s_v, x_v, t_v)$ be the set of all variables of node $v$ in order $(S, IV, \text{tmpS})$ used by Algorithm $\text{SORT}_c$ in the shared memory model.

Then $R$ is defined as follows. $R(S_1, \ldots S_n) = \{(S'_1, \ldots S'_n), t_i \in \{A, B\}, 1 \leq i \leq n\}$.

As one can see, for each configuration $C_1$ of Algorithm $\text{ASORT}_c$ in the abstract model, there exists $2^n$ configurations in $R(C_1)$ of Algorithm $\text{SORT}_c$ in the shared memory model, thus Condition (i) of Definition 2.1.1 is satisfied.

We are left to show that condition (ii) of Definition 2.1.1 is satisfied (Lemma 4.2).

**Lemma 4.2** Given $C_1$ and $C_2$, two configurations of Algorithm $\text{ASORT}_c$ in the abstract model, such that $C_1 \rightarrow C_2$ is an execution step of Algorithm $\text{ASORT}_c$; for any configuration $C'_1 \in R(C_1)$, if Algorithm $\text{SORT}_c$ 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 $\text{ASORT}_c$ in the abstract model in one chunk affects only the nodes' states in that chunk.

From Property 3.1.2 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 \( C_1 = (H_1^1, H_2^1, \ldots H_k^1) \) be the set of chunks, omitting the prefix of disabled nodes. Let \( v \) the first node in some chunk \( H_i^1 \) of Configuration \( C_1 \) of Algorithm \( ASORT_c \) in the abstract model. Assume without loss of generality that Action \( BAA \) is enabled at \( v \).

- If \( i = k \), \( H_k^1 \) 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 \( ASORT_c \) is:

\[
(A, x_v) \xrightarrow{BAA(v)} (B, x_v).
\]

In the shared memory model this corresponds to:

\[
((A, x_v), (A, y), (A, y), \ldots) \xrightarrow{BAA(v)} ((B, x_v), (A, y), (A, y), \ldots)
\]

that starts in any configuration of \( R(C_1) \) restricted to the chunk \( H_i^1 \), and ends in some configuration of \( R(C_2) \) restricted to the chunk \( H_i^1 \).

- If the chunk contains more than one node: \( H_i^1 = (S_v^1, S_{right,v}^1, S_2^1, \ldots S_n^1) \).

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

We have two cases, depending on whether \( v \) and \( right.v \) have an inversion.

1) nodes \( v \) and \( right.v \) do not have an inversion. Then the execution step of Algorithm \( ASORT_c \) is:

\[
((A, x),(s_1, y)) \xrightarrow{BAA(v)} ((B, x),(s_1, y)).
\]

In the shared memory model this corresponds to:

\[
((A, x, y), (A, y, y), (A, y, y), \ldots) \xrightarrow{BAA(v)} ((B, x, y), (A, y, y), (A, y, y), \ldots)
\]

that starts in any configuration of \( R(C_1) \) restricted to the chunk \( H_i^1 \), and ends in some configuration of \( R(C_2) \) restricted to the chunk \( H_i^1 \).

2) nodes \( v \) and \( right.v \) have an inversion. Then the execution step of Algorithm \( ASORT_c \) is:

\[
((A, x),(A, y), S_2^1, \ldots S_n^1) \xrightarrow{BAA(v)} ((B, y),(A, x), S_2^1, \ldots S_n^1).
\]

(\( v \) can affect only the variables of its right neighbor).

In the shared memory model this corresponds to:

\[
((A, x, y), (A, y, y), S_2^1, \ldots S_n^1) \xrightarrow{BAA(v)} ((X, x + y, B), (A, y, y), S_2^1, \ldots S_n^1) \xrightarrow{S2(right.v)} ((X, x + y, B), (Y, x, A), S_2^1, \ldots S_n^1) \xrightarrow{S3(right.v)} ((A, y, y), (A, x, y), S_2^1, \ldots S_n^1)
\]

that starts in any configuration of \( R(C_1) \) restricted to the chunk \( H_i^1 \), and ends in some configuration of \( R(C_2) \) restricted to the chunk \( H_i^1 \).
If the starting state of the node is either $A$ or $B$, then the value to be sorted is its initial value. If some node starting state is either $X$ or $Y$, then it is possible for some of the three steps of the swap to be applied (see Section 4.1.1) 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 4.3.1 shows that for any node $v$ whose state is $X$, 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 4.3.2 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 4.3.3 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 4.3).

**Property 4.3.1** 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.

**Proof.** Let $(s_0, x, t_0)$, respectively $(s_1, y, t_1)$, be the set of variables of node $v$, respectively $right.v$ (if $right.v \neq \perp$) in the order $(S, IV, tmpS)$. Since the values of variable $tmpS$ are drawn from the set $\{A, B\}$, we have that $t_0, t_1 \in \{A, B\}$, if they exist.

We analyze, by cases, what happens to the pair $(v, right.v)$ when Algorithm $SORT_c$ is executed locally. An execution step is marked by an arrow $\rightarrow$ labeled by a action that is enabled and has been selected for execution by the respective node. If more than one action is enabled at some node $v$, then we use split arrows to show possible execution steps. The underscore symbol _ means that the respective value does not matter.

We have the following cases:
1. \( \text{right.} v = \bot \). Then the only enabled action at node \( v \) is Action \( C2: ((X, \rightarrow t_0), \bot) \) \( \xrightarrow{C2(v)} \) \( ((t_0, \rightarrow \cdot), \bot) \).

2. \( \text{right.} v \neq \bot \land s_1 = Y \). Done.

3. \( \text{right.} v \neq \bot \land s_1 \in \{A, B\} \). Then node \( v \) is currently disabled and only Action \( S1 \) is enabled at \( \text{right.} v: ((X, \rightarrow \cdot), (s_1, \rightarrow \cdot)) \) \( \xrightarrow{S1(\text{right.} v)} \) \( ((X, \rightarrow \cdot), (Y, \rightarrow s_1)) \). Done.

4. \( \text{right.} v \neq \bot \land s_1 = X \).

Then only Action \( C3 \) is enabled at \( v \). Node \( \text{right.} v \) may have one of the following actions enabled: Action \( S2 \), Action \( C2 \) or Action \( C3 \). But the execution of these actions does not affect the variables of node \( v \). Also, since one cannot know what the values of the variables of node \( \text{right.} v \) will be after Action \( C3 \) is executed at node \( v \), we use the underscore symbol. 

\[
((X, \rightarrow t_0), (X, \rightarrow \cdot)) \xrightarrow{C3(v)} ((t_0, \rightarrow t_0), (\cdot, \rightarrow \cdot)).
\]

\( \square \)

**Property 4.3.2** 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 \( B \).

**Proof.** The property follows from the Action \( S2 \) of Algorithm \( \text{SORT}_c \). If \( S.v = X \land \text{right.} v \neq \bot \land S.\text{right.} v = Y \), then the only enabled action at \( v \) is Action \( S2 \), and node \( \text{right.} v \) is currently disabled. Once Action \( S2 \) executes in at most one round, \( S.v \) becomes either \( A \) or \( B \), and we are done. \( \square \)

**Property 4.3.3** If \( S.v = Y \), then in at most two rounds, \( S.v \) becomes either \( A \) or \( B \).

**Proof.** Let \((s_0, z, t_0)\), respectively \((s_2, z, t_2)\), be the set of variables of node \( v \), respectively \( \text{left.} v \) (if \( \text{left.} v \neq \bot \)). Again, \( t_0, t_2 \in \{A, B\} \).

We analyze, by cases, what happens to the pair \((\text{left.} v, v)\) when Algorithm \( \text{SORT}_c \) is executed. We have the following cases:
1. \( \text{left.}v = \bot \). Then \((\bot, (Y, \rightarrow t_0)) \xrightarrow{\text{C1(v)}} (\bot, (t_0, \rightarrow ))\), and we are done.

2. \( \text{left.}v \neq \bot \land s_2 \in \{A, B\} \). Then Action S3 is the only enabled action at node \( v \).
   Node \( \text{left.}v \) may have Action S1 enabled while Action S3 is executed at node \( v \). But
   since an execution of node \( \text{left.}v \) does not affect the variables of node \( v \), and since
   one cannot know what the values of the variables of node \( \text{left.}v \) will be after Action
   S3 is executed at node \( v \), we use the symbol underscore:
   \[ ((s_2, \rightarrow t_2), (Y, \rightarrow t_0)) \xrightarrow{\text{S3(v)}} ((\rightarrow \rightarrow \rightarrow), (t_0, \rightarrow \rightarrow \rightarrow)), \text{and we are done.} \]

3. \( \text{left.}v \neq \bot \land s_2 = X \). Node \( v \) is disabled until node \( \text{left.}v \) executes Action S2. Then
   by Property 4.3.3, node \( \text{left.}v \) executes Action S2 and the value of \( S.(\text{left.}v) \) change
to either A or B. We then apply case 2.

4. \( \text{left.}v \neq \bot \land s_2 = Y \). Only Action S3 is currently enabled at node \( v \). Node \( \text{left.}v \)
   may have Action S3 or Action C1 enabled while Action S3 will be executed at node \( v \). But since an execution of node \( \text{left.}v \) does not affect the variables of node \( v \), and since
   one cannot know what the values of the variables of node \( \text{left.}v \) will be after
   Action S3 is executed at node \( v \), we use the underscore symbol:
   \[ ((Y, \rightarrow \rightarrow), (Y, \rightarrow t_0)) \xrightarrow{\text{S3(v)}} ((\rightarrow \rightarrow \rightarrow), (t_0, \rightarrow \rightarrow \rightarrow)), \text{and we are done.} \]

\[ \square \]

**Lemma 4.3** For any node \( v \), if \( S.v \in \{X, Y\} \), in at most four rounds becomes \( S.v \) becomes
either A or B.

**Proof.** Directly from Properties 4.3.1, 4.3.2, and 4.3.3. \( \square \)

**Lemma 4.4** Starting from an arbitrary configuration, in at most \( 8n - 8 \) rounds, Algorithm
\( \text{SORT}_c \) arranges the \( n \) values in non-decreasing order from left to right starting from the
node \( L \) and going right.

**Proof.** From Lemma 4.3, each swap takes at most four rounds. From Lemma 4.1, if a
swap takes at most one round, then sorting takes at most \( 2n - 2 \) rounds. Since the swap
takes at most four rounds, we obtain a total of at most \( 8n - 8 \) rounds. \( \square \)
CHAPTER 5

SIMULTANEOUSLY ACTIVATED PROCESSES ON A RING

In this chapter we present Algorithm SSSSR (Section 5.1), followed by its proof of correctness (Section 5.1.1), and then give one application of the proposed algorithm: Algorithm CLRAR (Section 5.2). The other application of Algorithm SSSSR, Algorithm SORT_r, is presented in Chapter 6.

Algorithm SSSSR is a general self-stabilization scheme for simultaneously activated non-adjacent processes on an asynchronous oriented ring. It is semi-uniform and works under the weakly fair daemon. It is optimal in space complexity — n - 2 nodes use one bit each and the other two nodes use two bits each, where n is the total number of nodes. It is asymptotically optimal in the time complexity — for any t ≥ 0, every node is enabled at least t times within the first 3n/2 - 2 + 3t rounds, i.e., on the average, once every three rounds.

We then give two applications of the proposed algorithm: a space and time optimal solution to the local resource allocation problem (Algorithm CLRAR), and a space and asymptotically time optimal solution to the distributed sorting problem on the oriented ring where the values to be sorted are not necessarily distinct, and each process holds only one value at any moment (Chapter 6 - Algorithm SORT_r).

5.1 Self-Stabilizing Distributed Simultaneously Execution of Non-adjacent Nodes on an Oriented Ring SSSSR

Algorithm SSSSR is an extension of Algorithm SSSS to an oriented ring where the number of nodes is n ≥ 3, and it is presented in Figure 5.1. Every node holds a variable S ∈ {A, B}. Node L and its left (node IL = l_L) use another variable called lock ∈ {0, 1}
to "control" the link \((IL, L)\). Note that the node \(IL\) of the ring is similar to node \(R\) of the chain except that the nodes \(L\) and \(R\) are not connected directly in a chain. In a chain, both nodes \(L\) and \(R\) can be enabled at the same time. However, in a ring, either node \(L\) or \(IL\) can be enabled at any time, but not both.

For any node \(v\), let \(l_v, r_v\) be the left, respectively right neighbor of node \(v\), \(S = S.v, \) 
\(lock = lock.v, S_l = S.l_v, S_r = S.r_v\). Predicate \(link.ok(node \ v), v \in \{IL, L\}\), returns true is the link \((IL, L)\) allows node \(v\) to execute.

Macro \textit{execute} is a generic macro: the current node executes something based on its values and/or the values of its neighbors.

\begin{algorithm}
\algname{Self-Stabilizing Distributed Simultaneously Activating of Non-contiguous Nodes on an Oriented Ring SSVSTZ}
\begin{align*}
\text{Predicate } link.ok(v) &\equiv (v = L \land lock.IL \neq lock.L) \lor (v = IL \land lock.IL = lock.L) \\
\text{Actions for the ring leader node } L \\
BB &\quad S = B \land S_r = B \land link.ok(L) \quad \rightarrow \quad \text{execute}(v); \ lock = lock.IL; \ S = A \\
AA &\quad S = A \land S_r = A \land link.ok(L) \quad \rightarrow \quad \text{execute}(v); \ lock = lock.IL; \ S = B \\
CL &\quad ((S = A \land S_r = B) \lor (S = B \land S_r = A)) \land link.ok(L) \quad \rightarrow \quad lock = lock.IL \\
\text{Actions for the node } IL \\
AB &\quad S_l = A \land S = B \land link.ok(IL) \quad \rightarrow \quad \text{execute}(v); \ lock = 1 - lock.L; \ S = A \\
BA &\quad S_l = B \land S = A \land link.ok(IL) \quad \rightarrow \quad \text{execute}(v); \ lock = 1 - lock.L; \ S = B \\
CL &\quad (S_l = A \land S = A) \lor (S_l = B \land S = B) \land link.ok(IL) \quad \rightarrow \quad lock = 1 - lock.L \\
\text{Actions for any node } v \notin \{L, IL\} \\
ABB &\quad S_l = A \land S = B \land S_r = B \quad \rightarrow \quad \text{execute}(v); \ S = A \\
BAA &\quad S_l = B \land S = A \land S_r = A \quad \rightarrow \quad \text{execute}(v); \ S = B
\end{align*}
\end{algorithm}

A node other than the nodes \(L\) and \(IL\) is enabled to execute if the following two conditions
are true: (i) Its $S$ value is different from its left neighbor’s $S$ value, and (ii) its $S$ value is the same as its right neighbor’s $S$ value. Node $L$ is enabled to execute if the two lock values are different. Node $IL$ is enabled to execute if the two lock values are the same.

Actions $CL$ are called lock actions. The rest of the actions are non-lock actions.

Note that if either node $L$ or $IL$ is enabled, it is enabled to execute either a lock or a non-lock action, but never both. If node $L$ is enabled, it is enabled to execute a non-lock action if and only if its value is different from that of its right neighbor, otherwise a lock action. If node $IL$ is enabled, it is enabled to execute a non-lock action if and only if its value is the same as that of its left neighbor, otherwise a lock action.

All nodes other than the nodes $L$ and $IL$ execute the same actions as in Algorithm $SSDS$. As in Algorithm $SSDS$, nodes $IL$ and $L$ compare their $S$ value with the $S$ value of a single neighbor (right neighbor for node $L$ and left neighbor for node $IL$). In Algorithm $SSDSR$ they compare their lock values as described below:

- If node $L$ is not blocked by node $r_L$ (due to its $S$ value) and node $IL$ (due to its lock value), then it executes a non-lock action, changes its lock and $S$ values, and becomes disabled.

- If node $L$ is blocked by node $r_L$ (due to its $S$ value), but is not blocked by node $IL$ (due to its lock value), then it executes a lock action and changes its lock value (to unblock node $IL$).

The above rules are similar for node $IL$, as follows:

- If node $IL$ is not blocked by node $l_{IL}$ (due to its $S$ value) and node $L$ (due to its lock value), then it executes a non-lock action, changes its lock and $S$ values, and becomes disabled.

- If node $IL$ is blocked by node $l_{IL}$ (due to its $S$ value), but is not blocked by node $L$ (due to its lock value), then it executes a lock action and changes its lock value (to unblock node $L$).

For example, given a network of seven nodes (Figure 5.1(a)), the only enabled nodes are the odd numbered nodes starting with the leader (the leader, third, fifth, and seventh nodes).
If we assume a synchronous system, the next execution step brings the system into the configuration in Figure 5.1(b), and so on.

5.1.1 Proof of Correctness for Algorithm SSVSTZ

In this section we prove that Algorithm SSVSTZ stabilizes in at most \( 3n/2 - 2 + 3k \) rounds to the global predicate

\[
\text{k-Execute} \equiv \{ \forall \text{ node } v, v \text{ has executed macro } \text{execute} \text{ at least } k \text{ times } \}.
\]

We extend the terms of configuration-string and difference-string defined in Section 3.1.1 to oriented rings, and prove some properties of Algorithm SSVSTZ. We show that in every configuration:

- At most one node per neighborhood is enabled to execute a non-lock action (Property 5.1.2).
- There exists at least one node that is not blocked by its neighbors because of its S value (Property 5.1.3).
- Any node becomes disabled after executing its enabled action (Property 5.1.4).
- Either node L or node IL is enabled to execute (a lock or non-lock action) (Property 5.1.5).
- In any configuration in which \( S.L = S.right(L) \), node L will execute a non-lock action within two rounds (Property 5.1.6).
- In any configuration in which \( S.IL \neq S.left(IL) \), node IL will execute a non-lock action within two rounds (Property 5.1.7).

We then show that during the first \( 3n/2 - 2 + 3t \) rounds, every node is enabled at least \( t \) times, on the average, once every three rounds (Lemma 5.1).
We then show that Algorithm SSVSTZ works under the weakly fair distributed daemon. Since in every configuration there exists at least one enabled node (Property 5.1.5), we need only show that starting from any configuration where some node \( v \) is enabled, node \( v \) remains continuously enabled until it is selected to execute (Property 5.1.11).

Algorithm SSVSTZ does not work under the unfair distributed daemon, because in any configuration, either node \( L \) or node \( IL \) is enabled to execute (Property 5.1.5) but it works under the weakly fair daemon (Property 5.1.11).

We extend the definition of a configuration-string and difference-string to a ring as follows.

**Definition 5.1.1** A configuration-string of a \( n \)-node ring starts with the value of variable \( lock.L \), followed by the pound sign \( \# \), the value of variable \( S \) of all the nodes (starting from the leader node and going right), again the pound sign, and ending with the variable \( lock.IL \).

For example, the configuration-string of the configuration in Figure 5.1(a) is

\[ 1\#ABB\#\#\#\#\#\#P\#\#\#\#\#L\#P\#\#\#J\#L\#D\#Y\#M\#S\#G\#A\#I\#D\#\#\#L\#P\#A\#L\#E\#O\#J\#L \]

**Definition 5.1.2** Given \( C = lock_L\#S_LS_2\ldots S_IL\#lock_IL \) a configuration of an \( n \)-node ring, the difference-string \( b = b_{lock}\#b_1b_2\ldots b_{n-1}\# \) is obtained by appending to the lock difference bit \( b_{lock} \) (that is 0 if \( lock.IL = lock.L \), or 1 otherwise) the difference-string corresponding to \( S_LS_2\ldots S_IL \).

For example, the difference-string of the configuration in Figure 5.2(a) is \( 1\#10010101000010\# \) (Figure 5.2(b)).
Remark 5.1 Given a difference-string, some lock variable and some S variable of some node in a ring network, the corresponding configuration is uniquely defined.

Since any configuration of a ring has a unique configuration-string, and for any configuration-string there exists a unique configuration of a ring, from now one we use the term configuration instead of configuration-string.

Given a node v and a configuration C, let DSC(v) be the substring of DSC corresponding to its right neighbor (if any), itself, and its left neighbor (if any). For node L DSC(L) = b_{lock}#b_1, for node lL DSC(lL) = b_{n-1}#b_{lock}.

For example, given the configuration C from Figure 5.1(a), DSC(L) = 1#1, DSC(rL) = 10, and DSC(lL) = 0#1.

From the code of Algorithm SSVSTZ, we observe that:

Observation 5.1.1 Given any configuration C:

(i) Node L is enabled if and only if the lock difference bit b_{lock} = 1. Action BB or AA is enabled at node L if and only if DSC(L) is 1#0 and the execution of the action changes it from 1#0 → 0#1. Action CL is enabled at node L if and only if DSC(L) is 1#1 and the execution of the action changes it from 1#1 → 0#1 if node rL remains continuously disabled during this execution.

(ii) Node lL is enabled if and only if the lock difference bit b_{lock} = 0. Action AB or BA is enabled at node lL if and only if DSC(lL) is 1#0 and the execution of the action changes
Action $CL$ is enabled at node $lL$ if and only if $DS_C(lL)$ is $0\#0$ and the execution of the action changes it from $0\#0 \rightarrow 0\#1$ if node $l_{il}$ remains continuously disabled during this execution.

(iii) For $n > 2$, Action $ABB$ or $BAA$ is enabled at some node $v$ other than $L$ or $lL$, if and only if $DS_C(v)$ is $10$ and the execution of the action changes it from $10 \rightarrow 01$.

Property 5.1.2 For any node $v$, if node $v$ is enabled to execute a non-lock action, then neither node $l_v$ nor $r_v$ is enabled.

Proof. Depending on $v$, we have three cases:

1) $v = L$. Then, from Observation 5.1.1(i), $DS_C(v) = 1\#0$, thus $DS_C(r_v)$ starts with a $0$. From Observation 5.1.1(ii,iii), for node $r_v$ to be enabled to execute, then $DS_C(r_v) \in \{10,1\#0\}$, thus it should start with a $1$. Contradiction.

2) $v = lL$. Then, from Observation 5.1.1(ii), $DS_C(v) = 1\#0$, thus $DS_C(l_v)$ ends with a $1$. From Observation 5.1.1(i,iii), for node $l_v$ to be enabled to execute, then $DS_C(l_v) \in \{1\#0,1\#\}$, thus it should end with a $0$. Contradiction.

3) $v \notin \{lL, L\}$. Then, from Observation 5.1.1(iii), $DS_C(v) = 10$, thus $DS_C(l_v)$ ends with a $1$ and $DS_C(r_v)$ starts with a $0$. From Observation 5.1.1(i,iii), for node $l_v$ to be enabled to execute, then $DS_C(l_v) \in \{1\#0,1\#\}$, thus it should end with a $0$. Contradiction. From Observation 5.1.1(ii,iii), for node $r_v$ to be enabled to execute, then $DS_C(r_v) \in \{10,1\#0\}$, thus it should start with a $1$. Contradiction. □

Property 5.1.3 In any configuration $C$ at least one of the following conditions is true:

(i) node $L$ is not blocked by its right neighbor $r_L$

(ii) node $lL$ is not blocked by its left neighbor $l_{il}$

(iii) there exists at least one node $v$ other than $L$ and $lL$ that is enabled.

Proof. $DS_C = b_{lock}\#b_1 \ldots b_{n-1}\#$. We have three cases:

1) If $b_1 = 0$, then condition (i) is true. Done.

2) If $b_1 = 1$ and $DS_C$ contains the substring $10$, then by Observation 5.1.1(ii) some node $v$ is enabled, thus condition (iii) is true. Done.

3) If $b_{n-1} = 1$, then condition (ii) is true. Done. □
Property 5.1.4 For any node \( v \), if node \( v \) is enabled and the daemon selects it to execute, after the execution is completed, its actions are disabled.

Proof. From Observation 5.1.1(i, ii), if node \( v \) is either node \( L \) or \( lL \), after executing its enabled action, it becomes disabled. From Observation 5.1.1(iii), a node \( v \) other than node \( L \) or \( lL \) 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. \( \square \)

Property 5.1.5 In any configuration either node \( L \) or node \( lL \) is enabled to execute (a lock or non-lock action).

Proof. From Observation 5.1.1(i), node \( L \) is enabled if and only if \( block = 1 \). From Observation 5.1.1(ii), node \( lL \) is enabled if and only if \( block = 0 \). Since \( block \in \{0, 1\} \), either node \( l \) or node \( lL \) is enabled. \( \square \)

Property 5.1.6 If node \( rL \) does not block node \( L \) from executing, then it will execute a non-lock action within two rounds.

Proof. If node \( rL \) does not block node \( L \) from executing, and node \( L \) is still not enabled to execute a non-lock action, this means that the values \( lock.L = lock.lL \). From SSDSR's code, node \( lL \) is enabled to execute. It will do so in at most one round, and the values becomes \( lock.lL = 1 - lock.L \). Since node \( rL \) is disabled, in at most one round, node \( L \) is becomes enabled to execute a non-lock action. \( \square \)

Property 5.1.7 If node \( lL \) does not block node \( L \) from executing, then it will execute a non-lock action within two rounds.

Proof. Similar to the proof of Property 5.1.6. \( \square \)

To show that during the first \( 3n/2 - 2 + 3t \) rounds every node executes \( t \) times (Lemma 5.1), we need some additional notations, definitions and properties.

We extend the definition of a node not being blocked by another node from Section 3.1.1 to the ring where variable \( lock \) is also present as follows. Given two nodes \( v \) and \( r_v \) with \( S.v = a \) and \( S.r_v = b \), the notation "\( a \leftarrow b \)" denotes that state \( b \) does not block state \( a \) from
being enabled (in order for \( v \) being in state \( a \) to be enabled, \( S_v \) must be \( b \)). The notation \( a \rightarrow b \) indicates that state \( a \) does not block state \( b \) from being enabled (in order for \( v \) being in state \( b \) to be enabled, \( S_v \) must be \( a \)).

For nodes \( v = IL \) and \( v = L \) with lock\( .IL = a \) and lock\( .L = b \), by notation “\( a \leftarrow b \)” we denote that value \( b \) does not block value \( a \) from being enabled (in order for \( IL \) having lock\( .IL = a \) to be enabled, lock\( .L \) must be \( b \)). By notation \( a \rightarrow b \) we denote that value \( a \) does not block value \( b \) from being enabled (in order for \( L \) having lock\( .L = b \) to be enabled, lock\( .IL \) must be \( a \)). This parametric statement holds only because there are just two binary values to be compared.

For example the guard of Action \( AA \) can be re-written as \((0 \rightarrow 1, A \leftarrow A)\) or \((1 \rightarrow 0, A \leftarrow A)\), and the guard of Action \( AB \) can be re-written as \((A \rightarrow B, 1 \leftarrow 1)\) or \((A \rightarrow B, 0 \leftarrow 0)\).

Based on this notation, we adapt the sawtooth-like arrangement of nodes to be consistent with the difference-string of a ring configuration as follows. We start with the lock value of node \( IL \), going clockwise through the ring, and ending with the lock value of node \( L \). The difference-string of a given configuration is consistent with the orientation of the arrow between consecutive \( S \) values (1 for \( \nearrow \), 0 for \( \searrow \)).

For example, the sawtooth-like arrangement of the configuration in Figure 5.2(a) is given in Figure 5.3(a).

![Sawtooth-like notation and Delay values](image)

Figure 5.3: Configuration 1#BAAABBABABBABAA#0

We can then calculate the delay values of every node in the ring, as in Section 3.1.1. We start with value 0 for the node \( L \), decrease by 1 when going up, increase by 1 when going down, and in the end offset all the values by a positive integer such that the smallest delay
becomes 0. For example, the delay values for the nodes in configuration given in Figure 5.3(a) are depicted in Figure 5.3(b).

Nodes (iL) and (L) are abstract nodes, representing nodes L and iL. Their purpose is to correctly calculate the delay values for nodes L and iL when only the lock values are the ones to block some node from being enabled. When we discuss the delay value of node L or node iL, we ignore the delay values of the corresponding abstract node.

The delay value for a node represents the maximum number of rounds a node must wait until it becomes enabled to execute a non-lock action. For an n-node ring, the delay values are in the range 0...n.

After a node executes a non-lock action, the arrows to that node are reversed, and the delay values must be re-calculated. When node L or iL executes a lock action, only the arrow associated with the lock values is reversed.

The following properties from Section 3.1.1 are valid for the ring also.

**Property 5.1.8** In any configuration, if w is a neighbor of v such that (w, v) ≠ (iL, L) or (L, iL), then delay[w] = delay[v] ± 1.

**Proof.** From the definition of the delay values. □

**Property 5.1.9** For any t > 0:

(i) If S.v → S.r_v then node v cannot execute its enabled non-lock action for the t^{th} time until r_v has executed its enabled non-lock action for the t^{th} time, and node r_v cannot execute its enabled non-lock action for the (t+1)^{st} time until node v has executed its enabled non-lock action for the t^{th} time.

(ii) If S.v ← S.r_v then node r_v cannot execute its enabled non-lock action for the t^{th} time until node v has executed its enabled non-lock action for the t^{th} time, and node v cannot execute its enabled non-lock action for the (t+1)^{st} time until node r_v has executed its enabled non-lock action for the t^{th} time.

**Proof.** In case i), in order for node v to be enabled to execute a non-lock action, node r_v must change the orientation of the arrow between itself and node v. This will occur after node r_v is enabled to execute a non-lock action. By Property 5.1.4, after node r_v executes

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
an action (lock or non-lock action) for the \( t \)th time, it becomes disabled. Then for node \( r_v \) to become enabled again and to execute its enabled non-lock action for the \((t + 1)\)st time, node \( v \) must execute a non-lock action (for the \( t \)th time) and change the orientation of the arc toward node \( r_v \).

Case ii) is exactly opposite. □

Property 5.1.10 For any \( k > 0 \):

(i) If \( \text{lock.L} \rightarrow \text{lock.L} \) then node \( L \) cannot execute an action (lock or non-lock action) for the \( k \)th time until node \( L \) has executed an action for the \( k \)th time, and node \( L \) cannot execute an action for the \((k + 1)\)st time until node \( LL \) has executed an action for the \( k \)th time.

(ii) If \( \text{lock.L} \leftarrow \text{lock.L} \) then node \( L \) cannot execute an action for the \( k \)th time until node \( LL \) has executed an action for the \( k \)th time, and node \( LL \) cannot execute an action for the \((k + 1)\)st time until node \( L \) has executed an action for the \( k \)th time.

Proof. By Property 5.1.4, after node \( LL \) or \( L \) executes an action (lock or non-lock action) for the \( k \)th time, it becomes disabled. Then for node \( LL \) or \( L \) to become enabled again and to execute its enabled action for the \((k + 1)\)st time, node \( L \) respectively \( LL \) must execute an action (for the \( k \)th time) and change the orientation of the arc toward the other node. □

Let \( d_0 \) be the array of the delay values in the starting configuration, and \( D_0 \) be the maximum value in the array \( d_0 \). From Definition 3.1.3 of delay, \( 1 \leq D_0 \leq n \).

Lemma 5.1 For any node \( v \) and any value \( t > 0 \), within \( \frac{3d_0[v]}{2} + 3(t - 1) + 1 \) rounds node \( v \) executes a non-lock action at least \( t \) times.

Proof. We define the predicate \( P(q) \) as follows:

For any node \( v \), for any \( t \geq 1 \), node \( v \) executes a non-lock action \( t \) times within the first \( q \) rounds if:

- if node \( v \) is node \( L \) or \( LL \) then \( q > \frac{3}{2} d_0[v] + 3t - 3 \)
- if node \( v \) is not node \( L \) or \( LL \) then \( q \geq \frac{3}{2} d_0[v] + 3t - 3 \).

We prove by induction on \( q \geq 1 \) that Predicate \( P(q) \) 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 \( (t = 1) \).
Inductive step for $q > 1$, $\mathcal{P}(q - 1)$ and $\mathcal{P}(q - 2)$ hold. We have three cases:

1) Node $v = L$. We have that $q > \frac{3}{2}d_0[v] + 3t - 3$ and we must show that node $v$ executes a non-lock action $t$ times within the first $q$ rounds. From the induction hypothesis, we have that node $v$ has executed a non-lock action $t - 1$ times within the first $\frac{3}{2}d_0[v] + 3t - 6$ rounds.

We need to show that the right neighbor of $v$ does not block node $v$ from being enabled to execute a non-lock action for the $t^{th}$ time. Once the right neighbor does not block node $v$ from being enabled to execute a non-lock action, node $v$ will execute a non-lock action within the next two rounds: because it may be blocked by the lock value, by Property 5.1.6 node $v$ will execute a non-lock action within the next two rounds.

Let $u$ be the right neighbor of node $v$. We have two cases:

1.1) $d_0[u] = d_0[v] - 1$. Then $q > \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - \frac{1}{2} \geq \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 2 \geq \frac{3}{2}(d_0[v] - 1) + 3t - 3 \Rightarrow q - 2 \geq \frac{3}{2}d_0[u] + 3t - 3$. Since $\mathcal{P}(q - 2)$ is true for node $u$, this implies that node $u$ executed $t$ times within $q - 2$ rounds. From Property 5.1.9(i), node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q - 1$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

1.2) $d_0[u] = d_0[v] + 1$. Then $q > \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q \geq \frac{1}{2} + \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 2 \geq \frac{3}{2}(d_0[v] + 1) + 3t - 6 \Rightarrow q - 2 \geq \frac{3}{2}d_0[u] + 3t - 6$. Since $\mathcal{P}(q - 2)$ is true for node $u$, this implies that node $u$ executed $t - 1$ times within $q - 2$ rounds. From Property 5.1.9(ii), node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q - 1$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

2) Node $v = LL$. We have that $q > \frac{3}{2}d_0[v] + 3t - 3$ and we must show that node $v$ executes a non-lock action $t$ times within the first $q$ rounds. The proof is similar to the case 1) where instead of a right neighbor $u$ for node $L$ we have a left neighbor $u$ for node $LL$.

3) Node $v$ is neither node $L$ nor node $LL$. We have that $q \geq \frac{3}{2}d_0[v] + 3t - 3$ and we must show that node $v$ executes a non-lock action $t$ times within the first $q$ rounds. From the induction hypothesis, we have that node $v$ executed a non-lock action $t - 1$ times within the first $\frac{3}{2}d_0[v] + 3t - 6$ rounds. We need to show that neither the left neighbor of $v$, nor the right neighbor of $v$ blocks node $v$ from being enabled to execute a non-lock action for the $t^{th}$ time. If neither the right nor the left neighbor blocks node $v$ from being enabled to
execute a non-lock action, node $v$ will execute a non-lock action in the next round. Let $u$ be any neighbor of node $v$. We have two cases:

- $u = L$ or $IL$. Then $q \geq \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q + \frac{1}{2} > \frac{3}{2}d_0[v] + 3t - 3$. We have two cases:
  
  3.1) $d_0[u] = d_0[v] - 1$. Then $q + \frac{1}{2} > \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 1 > \frac{3}{2}(d_0[v] - 1) + 3t - 3 \Rightarrow q - 1 > \frac{3}{2}d_0[u] + 3t - 3$. Since $P(q - 1)$ is true for node $u$, this implies that node $u$ executes $t$ times within $q - 1$ rounds. From Property 5.1.9(i), node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

  3.2) $d_0[u] = d_0[v] + 1$. Then $q + \frac{1}{2} > \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 1 > \frac{3}{2}(d_0[v] + 1) + 3t - 6 \Rightarrow q - 1 > \frac{3}{2}d_0[u] + 3t - 6$. Since $P(q - 1)$ is true for node $u$, this implies that node $u$ executes $t - 1$ times within $q - 1$ rounds. From Property 5.1.9, node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

- $u \neq L$ and $IL$. We have two cases:

  3.3) $d_0[u] = d_0[v] - 1$. Then $q \geq \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 1 \geq \frac{3}{2}(d_0[v] - 1) + 3t - 3 \Rightarrow q - 1 \geq \frac{3}{2}d_0[u] + 3t - 3$. Since $P(q - 1)$ is true for node $u$, this implies that node $u$ executes $t$ times within $q - 1$ rounds. From Property 5.1.9, node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

  3.4) $d_0[u] = d_0[v] + 1$. Then $q \geq \frac{3}{2}d_0[v] + 3t - 3 \Rightarrow q - 1 \geq \frac{3}{2}(d_0[v] + 1) + 3t - 6 \Rightarrow q - 1 \geq \frac{3}{2}d_0[u] + 3t - 6$. Since $P(q - 1)$ is true for node $u$, this implies that node $u$ executes $t - 1$ times within $q - 1$ rounds. From Property 5.1.9(ii), node $u$ does not block node $v$ from being enabled for the $t^{th}$ time at the beginning of round $q$, so node $v$ will execute a non-lock action for the $t^{th}$ time within $q$ rounds.

\[\]

**Property 5.1.11** If some node $v$ is enabled to execute and it is not selected by the daemon to execute (it is not privileged), it remains enabled until it is selected.
Proof. We have two cases, depending on whether node $v$ is enabled to execute a lock or a non-lock action.

1) If node $v$ is enabled to execute a non-lock action, by Property 5.1.2 neither of the existing neighbors is enabled to execute a non-lock action. Thus node $v$ remains enabled until is selected.

2) If node $v$ is enabled to execute a lock action, by Property 5.1.5 the other neighbor (node $IL$ if $v = L$ or node $L$ if $v = IL$) is disabled. Node $v$ may become enabled to execute a non-lock action, but in any case it will remains enabled until selected. □

5.2 Self-Stabilizing Local Resource Allocation Algorithm on Oriented Rings \( \mathcal{LRAR} \)

Every node holds at most three variables. Two variables, variable $S$ that takes values in the set $\{A, B\}$, and Boolean variable $request$ that is true whenever the process requests access to its critical section $CS$, are common for all nodes. Node $L$ and its left (node $IL = II$) use another variable called $lock \in \{0, 1\}$ to "control" the link $(IL, L)$. For some node $v$, let $S = S.v$, $request = request.v$, and $lock = lock.v$. Predicate $link.ok(v)$ has been defined in Section 5.1.

5.2.1 Proof of Correctness of Algorithm \( \mathcal{LRAR} \)

In this section we show that Algorithm \( \mathcal{LRAR} \) solves the local resource allocation problem for an oriented ring.

Local Resource Allocation [CDP03] allows neighboring nodes to enter their critical section concurrently, provided they do not use conflicting resources.

An algorithm solves the LRA problem if it satisfies two conditions: (i) (resource conflict free – safety) two neighboring processes execute their critical sections simultaneously using resources $X$ and $Y$, respectively only if $X$ and $Y$ can be accessed concurrently by any neighboring processes. (ii) (no lockout – liveness) a process requesting to enter its critical section will eventually be granted.

Since at most one node per neighborhood is enabled to execute a non-lock action (Prop-
Algorithm 5.2.1 Algorithm LRAR

Actions for the ring leader node $L$

$BB \quad S = B \land S_r = B \land link-ok(L) \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} lock = lock.L
\hspace{1cm} $S = A$

$AA \quad S = A \land S_r = A \land link.ok(L) \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} lock = lock.L
\hspace{1cm} $S = B$

$CL \quad ((S = A \land S_r = B) \lor (S = B \land S_r = A)) \land link.ok(L) \rightarrow$ lock = lock.L

Actions for the node $IL$

$AB \quad S_i = A \land S = B \land link.ok(ll) \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} lock = 1 - lock.L
\hspace{1cm} $S = A$

$BA \quad S_i = B \land S = A \land link.ok(ll) \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} lock = 1 - lock.L
\hspace{1cm} $S = B$

$CL \quad ((S_i = A \land S = A) \lor (S_i = B \land S = B)) \land link.ok(ll) \rightarrow$ lock = 1 - lock.L

Actions for any node $v \notin \{L, IL\}$

$ABB \quad S_i = A \land S = B \land S_r = B \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} $S = A$

$BAA \quad S_i = B \land S = A \land S_r = A \rightarrow$
\hspace{1cm} if request then $CS; request = false$
\hspace{1cm} $S = B$

(Algorithms 5.1.2), Algorithm SSDDR allows only two pairs of nodes to execute simultaneously.

Nodes $L$ and $r_L$ may execute in the same time, but only when node $L$ executes a lock action and node $r_L$ executes a non-lock action. Nodes $IL$ and $l_IL$ may execute in the same time, but only when node $IL$ executes a lock action and node $l_IL$ executes a non-lock action.
Property 5.1.2 shows that Algorithm $\mathcal{CRAR}$ has the safety property, provided that the critical sections of processes $r_L$ and $l_L$ do not include any actions related to the variables $lock.L$, respectively $lock.lL$.

Lemma 5.1 shows that Algorithm $\mathcal{CRAR}$ has the liveness property.
In this chapter we present an application of Algorithm SSSR: the distributed sorting problem on the oriented ring with a distinguish node, where the values to be sorted are not necessarily distinct, and each process holds only one value at any moment.

We present two solutions: we first present a sorting algorithm for an abstract model of communication (Algorithm ASORT_r), then show how sorting will be done in the shared memory model of communication (Algorithm SORT_r).

Using some properties of Algorithm SSSR, we show that Algorithm ASORT_r stabilizes in at most $5n - 6$ rounds under the weakly fair distributed daemon (Section 6.2). Algorithm SORT_r reduces to Algorithm ASORT_r, and we conclude that Algorithm SORT_r stabilizes in at most $4(5n - 6)$ rounds (Lemma 6.2), thus it sorts $n$ values on a ring in non-decreasing order from left to right starting from the distinguished node in at most $4(5n - 6)$ rounds. It uses at most four bits per node. (Two distinguished nodes use four bits each, while the other nodes use three bits each.)

6.1 Self-Stabilizing Distributed Sorting Algorithms in an Oriented Ring

In this section we present two algorithms for distributed sorting problem in an oriented ring: ASORT_r (Section 6.1.1), and SORT_r (Section 6.1.2). Algorithm ASORT_r is implemented in an abstract model. Algorithm SORT_r is implemented in the shared memory model.

6.1.1 Distributed Sorting in an Oriented Ring

Each node, besides the variable $S$, holds one variable $IV$ to be sorted. Algorithm ASORT_r (Figure 6.1.2) is a particular case of Algorithm SSSR, 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. Only node IL does not swap its value with its right neighbor.

In the abstract model a node v, in order to execute the swap, can modify the right neighbor variable IV.r_v in certain situations. Intuitively, since by executing Algorithm SDDSR, local mutual exclusion is valid in any configuration (see Property 5.1.2), a node can synchronize the swap with its right neighbor, and once done, pass the token to it. We assume for now that the swap is done in an atomic step (macro swap), and we show in Section 6.1.2 how this is done in the shared memory model.

For some node v, let S = S.v, lock = lock.v, S_l = S.l_v, S_r = S.r_v. Predicate link.ok(v) has been defined in Section 5.1.

The sorting actions are mutually exclusive, so Algorithm ASORT_r is deterministic.

### 6.1.2 Sorting in the Shared Memory Model

In Algorithm SOTZTr (Figure 6.2), each node v holds three variables: variable IV to be sorted, variable S ∈ {A, B, X, Y}, and variable tmpS ∈ {A, B}. Variable tmpS stores the value of S temporarily while the swap is performed. Nodes rL and IL hold an extra variable lock.

Macro swap'(v, r_v, value) execute the first step of swapping between node v and its right node r_v, and variable tmpS.v stores value to be given to variable S.v after the swap is performed. Only node IL does not swap its value with its right neighbor. Predicate link.ok(v) has been defined in Section 5.1.

For some node v, let S = S.v, lock = lock.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.

The swapping is similar to the one for the chain network.

The synchronizing actions S1 – C1 are mutually exclusive with the sorting actions, and mutually exclusive among themselves, so Algorithm SOTZTr is deterministic also.
Algorithm 6.1.1 Self-Stabilizing Distributed Sorting in an Oriented Ring in the Abstract Model $A_{SO T Z}r$

Macro $\text{swap}(v,w) ::$ if $(w \neq \bot \land IV.v > IV.w)$ then
\[ IV.v = IV.v + IV.w; IV.w = IV.v - IV.w; IV.v = IV.v - IV.w \]

Actions for the leader node $L$

$BB$ $S = B \land S_r = B \land \text{link\_ok}(L) \rightarrow \text{swap}(v,r_v)$; $lock = lock.L$; $S = A$

$AA$ $S = A \land S_r = A \land \text{link\_ok}(L) \rightarrow \text{swap}(v,r_v)$; $lock = lock.L$; $S = B$

$CL$ $((S = A \land S_r = B) \lor (S = B \land S_r = A)) \land \text{link\_ok}(L) \rightarrow lock = lock.L$

Actions for the node $IL$

$AB$ $S_i = A \land S = B \land \text{link\_ok}(lL) \rightarrow lock = 1 - lock.L$; $S = A$

$BA$ $S_i = B \land S = A \land \text{link\_ok}(lL) \rightarrow lock = 1 - lock.L$; $S = B$

$CL$ $((S_i = A \land S = A) \lor (S_i = B \land S = B)) \land \text{link\_ok}(lL) \rightarrow lock = 1 - lock.L$

Actions for any node $v \notin \{L, IL\}$

$ABB$ $S_i = A \land S = B \land S_r = B \rightarrow \text{swap}(v,r_v)$; $S = A$

$BAA$ $S_i = B \land S = A \land S_r = A \rightarrow \text{swap}(v,r_v)$; $S = B$

6.2 Proof of Correctness of Algorithm $A_{SO T Z}r$

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

Definition 4.2.1 of pseudo-time is used for the ring topology as well.

For example, with the configuration and pseudo-time values in Figure 6.1(a), if the marked node executes then the next pseudo-time values are the ones in Figure 6.1(b).

Observation 4.2.1 and Corollary 4.2.2 are valid also for the ring.

The predicate $E(i,t)$ is changed as follows: there exists a configuration reachable from the initial configuration where the node at position $i$ is enabled with $\Psi(i) = t$, $1 \leq i \leq n$.

For the ring, we do not have a global constant PARITY.
Algorithm 6.2.1 Self-Stabilizing Distributed Sorting in an Oriented Ring in Shared Memory Model \textsc{SORT}_r

**Macro**\(\text{swap'}(v, w, tS) ::\) if \(w \neq 1 \land IV.v > IV.w\) then
\[
\text{tmpS.v} = tS; IV.v = IV.v + IV.w; S.v = X
\]

**Sorting Actions for the leader node** \(L\)

\(BB\) \(S = B \land S_r = B \land \text{link}.ok(L) \rightarrow \text{swap'}(v, r_v, A);\) lock = lock.\(L\)

\(AA\) \(S = A \land S_r = A \land \text{link}.ok(L) \rightarrow \text{swap'}(v, r_v, B);\) lock = lock.\(L\)

\(CL\) \((S = A \land S_r = B) \lor (S = B \land S_r = A)) \land \text{link}.ok(L) \rightarrow \text{lock} = \text{lock}.L\)

\(C1\) \(S = Y \rightarrow S = \text{tmpS}\)

**Actions for the node** \(L\)

\(AB\) \(S_l = A \land S = B \land \text{link}.ok(LL) \rightarrow \text{lock} = 1 - \text{lock}.L; S = A\)

\(BA\) \(S_l = B \land S = A \land \text{link}.ok(LL) \rightarrow \text{lock} = 1 - \text{lock}.L; S = B\)

\(CL\) \((S_l = A \land S = A) \lor (S_l = B \land S = B)) \land \text{link}.ok(LL) \rightarrow \text{lock} = 1 - \text{lock}.L\)

\(C1\) \(S = X \rightarrow S = \text{tmpS}\)

**Sorting Actions for any node** \(v \notin \{L, LL\}\)

\(ABB\) \(S_l = A \land S = B \land S_r = B \rightarrow \text{swap'}(v, r_v, A)\)

\(BAA\) \(S_l = B \land S = A \land S_r = A \rightarrow \text{swap'}(v, r_v, B)\)

**Synchronizing actions for any node** \(v \neq LL\)

\(S1\) \(S \in \{A, B\} \land S_l = X \rightarrow IV = IV_l - IV; \text{tmpS} = S ; S = Y\)

\(S2\) \(S = X \land S_r = Y \rightarrow IV = IV - IV_r ; S = \text{tmpS}\)

\(S3\) \(S = Y \land v \neq L \land S_l \neq X \rightarrow S = \text{tmpS}\)

\(C1\) \(S = X \land S_r = X \rightarrow S = \text{tmpS}\)

**Observation 6.2.1** (i) 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 \(k \geq 0\).

(ii) 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 \(k \geq 0\).
Given a starting configuration \( C_0 \) and \( C_j \) a configuration after Algorithm SSDS has executed a number of steps the relationship between the number of rounds that have elapsed and \( \Psi_j \) is given by Property 6.2.2.

**Property 6.2.2** Given a starting configuration \( C_0 \) and \( C_j \) some configuration after Algorithm SSDS has executed a number of steps, then the number of rounds elapsed \( q \leq 2 \times \min\{1 \leq i \leq n, \Psi_j(i)\} \).

**Proof.** A round has elapsed if all the enabled nodes have executed at least once. By Property 5.1.3, in every configuration there exists a node not blocked by its neighbors (or the neighbor, for nodes \( L \) and \( IL \)).

If node \( L \) (or \( IL \)) is one of the enabled nodes at the beginning of the round, then by Properties 5.1.6 or 5.1.7, it will execute a non-lock action (and increase its \( \Psi \) value by one unit) within two rounds.

If the enabled nodes are not \( L \) and \( IL \), then they increase their \( \Psi \) values by at least one unit within a round.

Thus the minimum value of \( \Psi \) increases by at least one every two rounds. \( \square \)

We use Definition 4.2.2 of array \( pos \) as in Section 4.2.

First, we show that the position of each element (array \( pos[] \)) is within a certain range depending on its pseudo-time value (Property 6.2.3). Then we show that once the \( \Psi \) values are at least \( 5n/2 - 3 \), the ring is sorted (Lemma 4.1).

**Property 6.2.3** For any rank \( r \), \( 1 \leq r \leq n \), and any value \( t \geq 0 \),

\[
pos[r, t] \leq \max \left\{ r, P[r, t] \right\} \quad \text{and} \quad \pos[r, t] \geq \min \left\{ r, Q[r, t] \right\}
\]
where \( P[r,t] = -t + 2r + 2n - 3 \) and \( Q[r,t] = t + 2r + 2 - 3n \), for any \( r \) and \( t \).

**Proof.** Consider the predicates:

\[
\begin{align*}
P(t) & : \text{for any rank } r, 1 \leq r \leq n, \quad \text{pos}[r,t] \leq \max \left\{ r \mid P[r,t] \right\} \quad \text{and} \\
Q(t) & : \text{for any rank } r, 1 \leq r \leq n, \quad \text{pos}[r,t] \geq \min \left\{ r \mid Q[r,t] \right\}
\end{align*}
\]

We show by induction on \( t \) that \( P(t) \) holds. The proof that \( Q(t) \) holds is similar.

*Base case: \( 0 \leq t \leq n - 1 \).* Since \( r \leq n \), then \( \max(r, P[r,t]) = P[r,t] \geq n \). From Definition 4.2.2, \( \text{pos}[r,t] \leq n \). Thus \( P(t) \) holds for any \( t, 0 \leq t \leq n - 1 \).

*Inductive step: \( t \geq n \).* \( P(t-1) \) is true, and we show that \( P(t) \) is true.

From the induction hypothesis, for any rank \( i < r \), \( \text{pos}[i,t-1] \leq \max \left\{ i \mid P[i,t-1] \right\} \).

Condition \( i < r \) implies that \( i \leq r - 1 \), so \( P[i,t-1] \leq P[r,t] - 1 \). We can conclude that

\[
\max \left\{ \frac{i}{i} \mid P[i,t-1] \right\} \leq \max \left\{ \frac{r-1}{r-1} \mid P[r,t] - 1 \right\} = \max \left\{ \frac{r}{r} \mid P[r,t] - 1 \right\}.
\]

Let

\[
M = \max \left\{ \frac{r}{r} \mid P[r,t] - 1 \right\}.
\]

Thus all the values whose ranks are less than \( r \) are in positions less or equal to \( M \) when their pseudo-times are \( t - 1 \), i.e. \( \forall i < r, \text{pos}[i,t-1] \leq M \).

And later on, their position cannot go beyond \( M \): \( \forall t' \geq t - 1, \text{pos}[i,t'] \leq M \), since there are no values less than \( r \) outside the position range \( 1 \ldots M \).

The element of rank \( r \) at pseudo-time \( t - 1 \) has the position \( \text{pos}[r,t-1] \leq M + 2 \):

By induction hypothesis, \( \text{pos}[r,t-1] \leq \max \left\{ \frac{r}{r} \mid P[r,t] + 1 \right\} \leq \max \left\{ \frac{r+1}{r+1} \mid P[r,t] + 1 \right\} = M + 2 \).

We show now that \( \text{pos}[r,t] \leq M + 1 \):

\[
\text{pos}[r,t] \leq M + 1 = \left\{ \frac{r+1}{r+1} \right\}, \text{ where } P[r,t] = -t + 2r + 2n - 3.
\]

If \( M + 1 \geq n \), the result is trivial. Assume that \( M + 1 < n \), which implies that \( t \geq n \), which implies that \( t - 2 \geq \Psi_0(M + 1) \) (Corollary 4.2.2).

At pseudo-time \( t - 1 \), the elements whose ranks are smaller than \( r \) are at positions less than \( M + 1 \), thus the ranks of the elements at positions \( M + 1 \) and \( M + 2 \) are at least \( r \).

We have two cases:
1) \( \text{pos}[r, t - 1] = M + 2 \). Thus there exists an element whose rank \( K > r \) such that \( \text{pos}[K, t - 1] = M + 1 \).

If \( E(M + 1, t - 1) \) holds then the node at position \( M + 1 \) will execute and swap the value \( K \) with \( r \), thus \( \text{pos}[r, t] = M + 1 \).

If \( E(M + 1, t - 1) \) is false, then \( E(M + 1, t - 2) \) holds. The node at position \( M + 1 \) has executed at pseudo-time \( t - 2 \) and as a result, \( \text{pos}[r, t - 1] = M + 2 \). Thus the value \( K \) at position \( M + 1 \) at pseudo-time \( t - 1 \) is smaller than \( r : K < r \). Contradiction.

2) \( \text{pos}[r, t - 1] = M + 1 \). Since the values at positions \( M + 2 \) and higher are larger than \( r \), the element of rank \( r \) cannot move to \( M + 2 \) in any of the subsequent times.

In both cases, \( \text{pos}[r, t] \leq M + 1 \), thus Predicate \( P(t) \) holds. \( \square \)

**Lemma 6.1** Algorithms \( \text{A.SORT}_r \) stabilizes in \( 5n - 6 \) rounds; thus the stabilization time is \( O(n) \) rounds.

**Proof.**

From Property 6.2.3 for \( t = 5n/2 - 3 \) and \( r \leq n/2 \), \( \text{pos}[r, 5n/2 - 3] \leq r \). For the node of rank 1, \( \text{pos}[1, 5n/2 - 3] \leq 1 \) and by Definition 4.2.2, \( \text{pos}[1, t] \geq 1 \forall t \geq 0 \). Thus \( \text{pos}[1, 5n/2 - 3] = 1 \). It follows that for all ranks \( r \leq n/2 \), \( \text{pos}[r, 5n/2 - 3] = r \).

From Property 6.2.3, for \( t = 5n/2 - 3 \) and \( r \geq n/2 + 1 \), \( \text{pos}[r, 5n/2 - 3] \geq r \). For the node of rank \( n \), \( \text{pos}[n, 5n/2 - 3] \geq n \) and by Definition 4.2.2, \( \text{pos}[n, t] \leq n \forall t \geq 0 \). Thus \( \text{pos}[n, 5n/2 - 3] = n \). It follows that for all ranks \( r \geq n/2 + 1 \), \( \text{pos}[r, 5n/2 - 3] = r \).

Thus for all ranks \( r \), \( \text{pos}[r, 5n/2 - 3] = r \).

By Property 6.2.2, after at most \( 2(5n/2 - 3) = 5n - 6 \) rounds, the values are sorted. \( \square \)

**6.2.1 Reduction of Algorithm \( \text{A.SORT}_r \) in Abstract Model to Algorithm \( \text{SORT}_r \) in Shared-Memory Model**

In this section we show that Algorithm \( \text{SORT}_r \) reduces to Algorithm \( \text{A.SORT}_r \).

Let \( S_L = (s_L, x_L, k_L) \), respectively \( S_{IL} = (s_{IL}, x_{IL}, l_{kIL}) \), be the set of all variables of nodes \( L \), respectively \( IL \), in the order \( (S, IV, lock) \), and \( S_v = (s_v, x_v) \) be the set of all variables of any other node \( v \) in the order \( (S, IV) \), used by Algorithm \( \text{A.SORT}_r \) in the abstract model. Let \( S_L^{IL} = (s_L, x_L, l_{kIL}, t_L) \) be the set of all variables of node \( L \), \( S_{IL}^{IL} = (s_{IL}, x_{IL}, l_{kIL}, t_{IL}) \) be
the set of all variables of node \( IL \) in the order \((S, IV, lock, tmpS)\), and \( S^v = (s_v, x_v, t_v) \) be the set of all variables of any other node \( v \) in the order \((S, IV, tmpS)\) used by Algorithm \( SORT_\tau \) in the shared memory model.

Then \( R \) is defined as follows. \( R(S_1, \ldots, S_n) = \{(S^t_1, \ldots, S^t_n), t_i \in \{A, B\}, 1 \leq i \leq n\} \).

As one can see, for each configuration \( C_1 \) of Algorithm \( ASORT_\tau \) in the abstract model, there exists \( 2^n \) configurations in \( R(C_1) \) of Algorithm \( SORT_\tau \) in the shared memory model, thus Condition (i) of Definition 2.1.1 is satisfied.

To show that Condition (ii) of Definition 2.1.1 is satisfied, we break the system configuration, starting from the leader node \( L \), into a number of chunks. The proof follows similarly to the reduction proof of Algorithm \( SORT_\tau \) to Algorithm \( ASORT_\tau \).

We can then conclude that, starting from an arbitrary configuration, in at most \( 20n - 24 \) rounds, Algorithm \( SORT_\tau \) sorts the values in non-decreasing order (Lemma 6.2).

**Lemma 6.2** Starting from an arbitrary configuration, in at most \( 20n - 24 \) rounds, Algorithm \( SORT_\tau \) arranges the \( n \) values in non-decreasing order from left to right.

**Proof.** Every swap takes at most four rounds. From Lemma 6.1, if a swap takes at most one round, then sorting takes at most \( 5n - 6 \) rounds. Since the swap takes at most four rounds, we obtain a total of at most \( 20n - 24 \) rounds. \( \square \)
CHAPTER 7

SIMULTANEOUSLY ACTIVATED PROCESSES ON A TREE

In this chapter we present Algorithm \textit{SSDST} (Section 7.1), followed by its proof of correctness (Section 7.1.1), and then give one application of the proposed algorithm: Algorithm \textit{LMET} (Section 7.2).

The other application of Algorithm \textit{SSDST}, Algorithm \textit{HEAP}, is presented in Chapter 8, Section 8.1.

Algorithm \textit{SSDST} is a general self-stabilization scheme for simultaneously activated non-adjacent processes on an asynchronous rooted tree. It is uniform and works under the unfair distributed daemon. It is optimal in space complexity; it uses $[\log(\text{deg})]$ bits in each node ($\text{deg}$ is the node degree). It is asymptotically optimal in time complexity: during the first $2h + 2t - 1$ rounds, every node is enabled at least $t$ times, i.e., on the average, once every second round. For a synchronous system, after at most $2h$ steps, every node is enabled every second step. If the synchronous network starts in a normal starting configuration, then a node is active every other step from the beginning.

We then give two applications on rooted trees of the proposed algorithm: a time and space optimal solution to the local mutual exclusion problem (Algorithm \textit{LMET}), and a space and (asymptotic) time optimal solution to the heap problem (Chapter 8 - Algorithm \textit{HEAP}).

7.1 \hspace{1em} Self-Stabilizing Distributed Simultaneous Execution of Non-adjacent Nodes in a Rooted Tree \textit{SSDST}

We extend the chain algorithm Algorithm \textit{SSDS} (Chapter 3) to a rooted tree in the following manner. A node is enabled to execute if the following two conditions are true:
(i) either it has no parent, or its parent’s state is different from its state, and (ii) all its children's states are the same as its state.

For simplicity we write $S = S.v$.

The predicate $\text{check}(v)$ has as parameter a node ID and is responsible for checking whether the given node $v$ exists, and if so, whether it has a certain value for its variable $S$. Macro $\text{execute}(v)$ is a generic macro: node $v$ executes something based on its values and the values of its neighbors.

**Algorithm 7.1.1** Algorithm SSDST

**Predicate** $\text{check}(v, s) \equiv (v = 1 \lor S.v = s)$

**Actions for any node $v$**

- **ABB** $S = B \land \text{check}(p_v, A) \land \forall w \in D_v : \text{check}(w, B) \rightarrow \text{execute}(v) ; S = A$
- **BAA** $S = A \land \text{check}(p_v, B) \land \forall w \in D_v : \text{check}(w, A) \rightarrow \text{execute}(v) ; S = B$

For example, given a network of eight nodes starting in a normal starting configuration (Figure 7.1(a)), the only enabled nodes are of even depth (the root and the children of the root’s children). If we assume a synchronous system, the next execution step brings the system into the configuration in Figure 7.1(b), in which the only enabled nodes are of odd depth. The next configuration is shown in Figure 7.1(c), followed by the one in Figure 7.1(d). Then the system returns to the configuration illustrated in Figure 7.1(a). The cycle repeats forever.
7.1.1 Proof of Correctness for Algorithm $SSDST$

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

\[ k-\text{Exec} \equiv \{ \forall \text{ node } v, v \text{ has executed macro execute at least } k \text{ times} \} \]

and works under the unfair distributed daemon.

We extend the notions of configuration-string and difference-string to the tree network, and we prove some properties of Algorithm $SSDST$. We then show that Algorithm $SSDST$ works under the unfair distributed daemon (Property 7.1.9, Section 7.1.2).

We then show that in every configuration, by executing Algorithm $SSDST$:

- Only one node per neighborhood is enabled (local mutual exclusion) (Property 7.1.2)
- At least one node is enabled in the tree (no deadlock) (Property 7.1.4)
- After it executes, a node becomes disabled until all its neighbors execute (fairness) (Property 7.1.3)
- During the first $2h + 2t - 1$ rounds every node executes at least $t$ times (no starvation and 1-fairness) (Lemma 7.1).

If $n = 1$ (unique node) and its starting state is $A$, then the node executes alternatively Action $BAA$, followed by $ABB$, forever. Thus we can assume for the rest of the proof that $n > 1$.

Let the configuration tree be the tree in which every node is represented by its variable $S$ only.

We call a normal starting configuration a configuration in which each branch of the configuration tree is a prefix of the $(AABB)^n$ (the string of length $4n$ obtained by concatenating...
AABB \( n \) times). Starting from a normal starting configuration, the enabled nodes are of even depth (Figure 7.1).

Each edge in the tree can be labeled by a bit depending on whether the adjacent nodes have or not the same \( S \) value (we call this a binary edge labeling).

**Definition 7.1.1** Given some configuration tree \( C \), we let \( DT_C \), the difference tree be the tree in which every node \( v \) is represented by a two-bit string \( DT_C(v) = b_0b_1 \) such that:

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

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

For example, for the configuration in Figure 7.1(a), the binary edge labeling is given in Figure 7.2(a) and the difference tree is given in Figure 7.2(b).

![Binary edge labeling](a) ![Difference tree](b)

**Figure 7.2: Example of a tree with eight nodes**

**Remark 7.1** (i) Given a binary edge labeling and the value \( S \) of some node, the corresponding configuration tree \( C \) is uniquely defined.

(ii) Given a difference tree \( DT \) and the value \( S \) of some node, the corresponding configuration tree \( C \) is uniquely defined.

**Observation 7.1.1** Given any configuration tree \( C \), a node \( v \) is enabled if and only if \( DT_C(v) = 10 \).

**Proof.** Given any configuration-tree \( C \), we have three cases, based on node \( v \).

**Case 1)** If \( v = R \) then by Definition 7.1.1, the first bit of \( DT_C(R) \) is 1. Guard \( ABB \) or \( BAA \) is enabled at node \( R \) if and only if all the children of node \( R \) have the same \( S \) value.
as \( R \), which is equivalent to all the links incident to \( R \) are labeled 0. By Definition 7.1.1, that corresponds to the second bit of \( DT_C(R) \) to be 0. Thus \( DT_C(R) \) has to be 10 in order for node \( R \) to be enabled and vice-versa.

Case 2) If \( v \) is a leaf node then by Definition 7.1.1, the second bit of \( DT_C(v) \) is 0. Guard \( ABB \) or \( BAA \) is enabled at node \( v \) if and only if the parent node \( p_v \) has a different \( S \) value, which is equivalent to the link \( (p_v, v) \) is labeled 1. By Definition 7.1.1, that corresponds to the the first bit of \( DT_C(v) \) to be 1. Thus \( DT_C(v) \) has to be 10 in order for node \( v \) to be enabled and vice-versa.

Case 3) If \( v \) is an internal node then Guard \( ABB \) or \( BAA \) is enabled at node \( v \) if and only if the parent node \( p_v \) has a different \( S \) value and all the children have the same \( S \) value as node \( v \), which is equivalent to the link \( (p_v, v) \) is labeled 1 and all the links \( (v, child[j]_v) \) \( 0 \leq j \leq D_v - 1 \) are labeled 0. By Definition 7.1.1, that corresponds to \( DT_C(v) = 10 \). Thus \( DT_C(v) \) has to be 10 in order for node \( v \) to be enabled and vice-versa. □

If the configuration tree \( C \) is understood, then we write \( DT \) instead of \( DT_C \).

**Property 7.1.2** For any configuration tree \( C \) and for any node \( v \), if node \( v \) is enabled to execute, then no neighbor of \( v \) is enabled.

**Proof.** Node \( v \) is enabled if and only if \( DT(v) = 10 \) (Observation 3.1.1).

By Definition 7.1.1, if node \( v \) has a parent \( p_v \) then \( DT(p_v) \) has the second bit 1: \( DT(p_v) \in \{01, 11\} \), thus by Observation 3.1.1 node \( p_v \) cannot be enabled.

By Definition 7.1.1, if node \( v \) has a child \( child[j]_v \) then \( DT(child[j]_v) \) has the first bit 0: \( DT(child[j]_v) \in \{01, 00\} \), thus by Observation 3.1.1 node \( child[j]_v \) cannot be enabled. □

**Property 7.1.3** 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.** By Observation 3.1.1 if node \( v \) is enabled then \( DT(v) = 10 \). If node \( v \) executes, then \( DT(v) \) changes as follows:

(i) If \( v = R \) then after execution of node \( R \) \( DT(R) \) becomes 11.

(ii) If \( v \) is a leaf node then after execution of node \( v \) \( DT(v) \) becomes 00.

(iii) If \( v \) is an internal node then after execution of node \( v \), \( DT(v) \) becomes 01.
In any case, after execution node $v$ is disabled (Observation 3.1.1).

Property 7.1.4 In any configuration tree $C$ there exists at least one enabled node.

Proof. If $DT(R) = 10$, we are done.

Otherwise, $DT(R)$ must be 11. By Definition 7.1.1, this implies that there exists a maximal length path starting from the root $R$ and ending at some node $v \neq R$ such that every link on the path is labeled 1. We have two cases:

Case 1) If node $v$ is a leaf, then by Definition 7.1.1 we obtain that $DT(v) = 10$. Thus by Observation 3.1.1 node $v$ is enabled.

Case 2) If node $v$ is an internal node, then by Definition 7.1.1 and the fact that the path has the maximal length we obtain that $DT(v) = 10$. Thus by Observation 3.1.1 node $v$ is enabled.

We use the definition of a node not being blocked by another node from Chapter 3 for the tree, described below. Given a node $v$ and its parent $p_v$ where $S.p_v = a$ and $S.v = b$, the notation "$a \leftarrow b$" denotes that state $b$ does not block state $a$ from being enabled (for $p_v$ to be enabled in state $a$, $S.v$ must be $b$). The notation $a \rightarrow b$ indicates that state $a$ does not block state $b$ from being enabled (for $v$ to be enabled in state $b$, $S.p_v$ needs to be $a$).

We use the above notation to define layers as follows.

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

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

We can represent a configuration tree using this notation in a level ordering, where the peak nodes are the enabled nodes. The binary edge labeling is consistent with the orientation of the arrows between a node and its parent, and a node and its children (1 for $\rightarrow$, 0 for $\leftarrow$). For example, the sawtooth-like arrangement of the configuration tree in Figure 7.3(a) is given in Figure 7.3(b).

We use Definition 3.1.3 of node delay from Chapter 3.
For a tree of height $h$, for any node $v$, $\text{delay}[v]$ is a value between 0 and $2h$. The number of rounds that a node waits before it becomes enabled cannot exceed its delay value.

The delay values of the nodes in Figure 7.3(a) are given in Figure 7.3(c). An enabled node has all the adjacent arrows pointing towards it. After an enabled node executes, the arrows are reversed and the delay values must be recalculated.

Property 7.1.5 In any configuration, if $w$ is a neighbor of $v$ then $\text{delay}[w] = \text{delay}[v] \pm 1$.

Property 7.1.6 For any $t > 0$:

(i) If $S.p_v \rightarrow S.v$ then node $p_v$ 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 $p_v$ has executed its enabled guard for the $t$th time.

(ii) If $S.p_v \leftarrow S.v$ then node $v$ cannot execute its enabled guard for the $t$th time until node $p_v$ has executed its enabled guard for the $t$th time, and node $p_v$ cannot execute its enabled guard for the $(t + 1)^{st}$ time until node $v$ has executed its enabled guard for the $t$th time.

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

Lemma 7.1 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 $q \geq d_0[v] + 2t - 1$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
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 some neighbor of node \( v \) (the parent or any child). From Property 7.1.5, \( d_0[u] = d_0[v] \pm 1 \). 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 Property 7.1.6, 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 Property 7.1.6, node \( u \) does not block node \( v \) from being enabled for the \( t^{th} \) time during round \( q \).

Neither the parent nor the children of node \( v \) 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 7.1.7** For any node \( v \) and any value \( t > 0 \) node \( v \) executes \( t \) times within the first \( 2h + 2t - 1 \) rounds.

**Proof.** Follows from Lemma 7.1: for any node \( v \), \( 2h \geq d_0[v] \). \( \square \)

### 7.1.2 The Unfair Distributed Daemon

In this section we show that Algorithm SSDST works under the unfair distributed daemon. A sufficient condition to prove that a certain algorithm works under the unfair
daemon is to show that a continuously enabled node which is never selected eventually becomes the only enabled node. If a node $v$ is enabled to execute but not selected by the distributed daemon, it remains enabled (Property 7.1.8). 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 7.1.9).

**Property 7.1.8** 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 7.1.2, neither of the existing neighbors is enabled. Since $v$ is not selected by the daemon to execute, the neighboring nodes remain disabled until $v$ gets selected. □

**Property 7.1.9** Every continuously enabled node will be eventually selected by the unfair distributed daemon.

**Proof.** By contradiction. Assume that there exists a node $v$ in the tree that is continuously enabled but the unfair daemon never selects it for execution. Since the executions of Algorithm $SSDST$ are infinite, starting from any arbitrary state, then there exists at least one node $u$ such that $u$ is executed infinitely often. Let $A$ be the maximal set of nodes in the tree that execute infinitely often, and suppose $v \notin A$.

If node $u$ executes infinitely often, then both neighbors of $u$ execute infinitely many often (Property 7.1.3, Lemma 7.1). Thus, if $u \in A$, then $left(u), right(u) \in A$. By induction, $A$ consists of all nodes. Contradiction. □

### 7.2 Self-Stabilizing Local Mutual Exclusion Algorithm on Rooted Trees $LMET$

In this section we present Algorithm $LMET$, followed by its proof of correctness. Each node holds two variables: variable $S$ that takes values in the set $\{A, B\}$, and Boolean variable $request$ that is $true$ whenever the process requests access to its critical section $CS$. For some node $v$, let $S = S.v$ and $request = request.v$. Predicate $check(v)$ has been defined in Section 7.1.
Algorithm 7.2.1 Algorithm $\mathcal{LME}_T$

\begin{align*}
A_{BB} & \quad S = B \land \text{check}(p, A) \land \forall w \in D_w : \text{check}(w, B) \\
& \quad \quad \quad \quad \rightarrow \text{if request then } CS; \text{ request } = false \\
& \quad \quad \quad \quad \quad S = A \\
B_{AA} & \quad S = A \land \text{check}(p, B) \land \forall w \in D_w : \text{check}(w, A) \\
& \quad \quad \quad \quad \rightarrow \text{if request then } CS; \text{ request } = false \\
& \quad \quad \quad \quad \quad S = B
\end{align*}

7.2.1 Proof of Correctness of Algorithm $\mathcal{LME}_T$

In this section we show that Algorithm $\mathcal{LME}_T$ solves the local mutual exclusion problem for a rooted tree.

A protocol solves the local mutual exclusion problem if any configuration of the system running the protocol has two properties ([AN02]): (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 7.1.2 shows that $\mathcal{LME}_T$ has the safety property. Lemma 7.1 shows that $\mathcal{LME}_T$ has the liveness property.
CHAPTER 8

HEAP CONSTRUCTION

In this chapter we present three heap algorithms.

The first two algorithms are applications of Algorithm $SSDST$, where the values to be sorted are not necessarily distinct, and each process holds only one value at any moment. We first present a min-heap algorithm for an abstract model of communication (Algorithm $A\_HEAP$ – Section 8.1.1), then show how min-heap construction will be done in the shared-memory model of communication (Algorithm $HEAP$ – Section 8.1.2).

Using some properties of Algorithm $SSDST$, we show that Algorithm $A\_HEAP$ stabilizes in at most $7h/2 - 4$ rounds under the unfair distributed daemon (Section 8.2). We then show that Algorithm $HEAP$ reduces to Algorithm $A\_HEAP$ (Section 8.3), thus it arranges $n$ values as a min-heap in at most $4(7h/2 - 4)$ rounds.

The third algorithm applies to rooted binary trees in which each node holds a number of items independent on the number of nodes in the tree (Algorithm $Heap$ – Section 8.4). It is the first snap-stabilizing distributed solution to the heap problem.

8.1 Self-Stabilizing Min-Heap Algorithms for a Rooted Tree

In this section we present two algorithms for min-heap problem in a rooted tree: $A\_HEAP$ (Section 8.1.1), and $HEAP$ (Section 8.1.2). Algorithm $A\_HEAP$ is implemented in an abstract model. Algorithm $HEAP$ is implemented in the shared-memory model.

8.1.1 Heap Construction in a Rooted Tree

Each node, besides the variable $S$, holds one variable $IV$ to be sorted. Algorithm $A\_HEAP$ (Figure 8.1.1) is a particular case of Algorithm $SSDST$, in which the macro $execute(v)$ is replaced by the macro $heap(v)$ that selects in $IV.v$ the minimal value among
itself and its children.

Consider an abstract model, different from the shared-memory model, in which a node
v, in order to have the heap property locally, can modify the variable \( IV.w \) of some child
\( w \). Intuitively, since by executing Algorithm \( SSDST \), local mutual exclusion is satisfied in
any configuration (see Property 7.1.2), a node can synchronize the swap of values with some
child. We assume for now that the swap is done in one atomic step (macro heap), and we
show in Section 8.1.2 how this is done in the shared-memory model.

For some node \( v \), let \( S = S.v \) and \( IV = IV.v \). Predicate \( check(v) \) has been defined in
Section 7.1. If all children of \( v \) hold values greater than or equal to \( IV \), then \( min(v) \) returns
the default value \(-1\). Otherwise, \( min(v) \) returns the index in the array \( D_v \) of a child of
node \( v \) which holds the minimum value.

---

**Algorithm 8.1.1 Self-Stabilizing Min-Heap in a Rooted Tree in the Abstract Model**

**Algorithm A.Heap**

**Macro heap(v) ::**

\[
\begin{align*}
    j &= \text{min}(v) \\
    \text{if } (j \geq 0 \land IV.v > IV.child[j].v) \text{ then} \\
    w &= \text{child}[j] \\
    IV.v &= IV.v + IV.w \\
    IV.w &= IV.v - IV.w \\
    IV.w &= IV.v - IV.w
\end{align*}
\]

**Function min(v) ::**

\[
\begin{align*}
    &\text{if } D_v = \bot \text{ then return } -1 \\
    \text{else} \\
    &\quad j = 0 \\
    \quad \text{forall } l \in \{0, |D_v| - 1\} \\
    \quad \text{if } (IV.child[j].v < IV.child[l].v) \text{ then } j = l \\
    \quad \text{if } (IV.child[j].v < IV.v) \text{ then return } j \\
    \quad \text{else return } -1
\end{align*}
\]

**Heap actions for any node \( v \)**

\[
\begin{align*}
    ABB & \quad S = B \land \text{check}(p_v, A) \land \forall w \in D_v : \text{check}(w, B) \quad \rightarrow \quad \text{heap}(v); S = A \\
    BAA & \quad S = A \land \text{check}(p_v, B) \land \forall w \in D_v : \text{check}(w, B) \quad \rightarrow \quad \text{heap}(v); S = B
\end{align*}
\]
The heap actions are mutually exclusive, so Algorithm $A\cdot\mathcal{HEAP}$ is deterministic.

### 8.1.2 Heap Construction in the Shared-Memory Model

In Algorithm $\mathcal{HEAP}$ (Figure 8.1.2), each node $v$ holds four variables: variable $IV$ to be sorted, variable $S \in \{A,B,X,Y\}$, variable $J$, and variable $tmpS \in \{A,B\}$. Variable $tmpS$ stores the value of variable $S$ temporarily while the swap is performed between node $v$ and its child $J$.

For any node $v$, let $S = S.v$, $IV = IV.v$, $J = J.v$, $tmpS = tmpS.v$, $S_p = S.p_v$, $J_p = J.p_v$, $IV_p = IV.p_v$, $S_J = S.(J.v)$, and $IV_J = IV.(J.v)$. The macro $heap'(v, value)$ executes the first step of swapping between node $v$ and the child $child[J].v$, 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 7.1. Function $min(v)$ has been defined in Section 8.1.1.

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

In Figure 8.1, nodes $v$ and $J$ need to swap their values. The state of each node is a triple $S;IV;tmpS$.

\[v: A;S_v: heap'(v,\text{true}); X:6;B \]
\[w: A;J_v: 5;6;Y:5;A \]
\[730; B:1;5;\]

**Figure 8.1:** Nodes $v$ and $J$ swap their $IV$ values

The synchronizing actions $S1-C3$ are mutually exclusive with the sorting actions, and mutually exclusive among themselves, so Algorithm $\mathcal{HEAP}$ is deterministic also.
Algorithm 8.1.2  \textit{Self-Stabilizing Heap in a Rooted Tree in the Shared-Memory Model } \\ \textit{HEAP}

\textbf{Macro} \texttt{heap'(v, tS)} ::
\begin{align*}
  j &= \min(v) \\
  \text{if} \ (j \geq 0 \land IV.v > IV.\text{child}[j]v) \text{ then} \\
  J &= \text{child}[j]v \\
  \text{tmpS}.v &= tS \\
  IV.v &= IV.v + IV.J \\
  S.v &= X
\end{align*}

\textbf{Heap actions for any node } v

\textit{ABB } \quad S = B \land \text{check}(p_v, A) \land \forall w \in D_v : \text{check}(w, B) \quad \longrightarrow \quad \texttt{heap'(v, A)}

\textit{BAA } \quad S = A \land \text{check}(p_v, B) \land \forall w \in D_v : \text{check}(w, A) \quad \longrightarrow \quad \texttt{heap'(v, B)}

\textbf{Synchronizing actions for any node } v

\begin{align*}
  s_1 & \quad S \in \{A, B\} \land p_v \neq \bot \land S_p = X \land J_p = v \quad \longrightarrow \quad IV = IV_p - IV ; \text{tmpS} = S ; S = Y \\
  s_2 & \quad S = X \land J \neq \bot \land S_J = Y \quad \longrightarrow \quad IV = IV - IV_J ; S = \text{tmpS} \\
  s_3 & \quad S = Y \land p_v \neq \bot \land S_p \neq X \quad \longrightarrow \quad S = \text{tmpS} \\
  c_1 & \quad S = Y \land p_v = \bot \quad \longrightarrow \quad S = \text{tmpS} \\
  c_2 & \quad S = X \land D_v = \bot \quad \longrightarrow \quad S = \text{tmpS} \\
  c_3 & \quad S = X \land D_v \neq \bot \land \exists w \in D_v : S.w = X \quad \longrightarrow \quad S = \text{tmpS}
\end{align*}

8.2 Proof of Correctness of Algorithm A.\textit{HEAP}

The root node R has level 1.

Besides local mutual exclusion, sorting requires synchronization between neighboring nodes. Each node has a local clock measuring \textit{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.

We adapt the \textit{pseudo-time} function from Chapter 4 to the tree; for each configuration, the \textit{pseudo-time} function \( \Psi \) is defined from the node to non-negative integers. \( \Psi \) is initially computed from the delay values, and is updated at each step. This function is computed recursively from the previous configuration, starting with the initial configuration.

96
ψ₀, the pseudo-time at the initial configuration, is defined as follows: (i) given node
and its parent \( p_v \), \( \psi_0(v) = \frac{d_0[v] + d_0[p_v]}{2} - 1 \), and (ii) the \( \psi_0 \) value of the root
node \( R \) is equal to the maximal \( \psi_0 \) value of its children \( \psi_0(R) = \max\{\psi_0(v), v \in \text{child}_R\} \), where \( R \) is the root.

For example, given the configuration in Figure 7.3(c), the \( \psi_0 \) values are given in Figure 8.2(a).

We observe that if a node \( v \) is enabled, then \( \psi_0(v) = \psi_0(w) \) for all \( w \in \text{D}_v \).

We adapt the definition of pseudo-time function from Chapter 4 to the tree. Function
\( \Psi \) is defined over the set of nodes (in Chapter 4 the function \( \Psi \) was defined over the set of
positions).

**Definition 8.2.1** Let \( \Psi_j \) and \( \Psi_{j+1} \) be the pseudo-time functions for two consecutive con­
figurations in some execution \( C_j \mapsto C_{j+1} \). Then \( \Psi_{j+1} \) is computed as follows:

- if node \( v \) has executed during this step then \( \Psi_j(v) \) and \( \Psi_j(w) \) for all children \( w \in \text{D}_v \)
increase by 1: \( \Psi_{j+1}(v) = \Psi_j(v) + 1 \) and \( \Psi_{j+1}(w) = \Psi_j(w) + 1 \).

- if any child of the root \( R \) executes, \( \Psi(R) \) is updated if necessary, i.e.,

\[
\Psi_{j+1}(R) = \max_{w \in \text{child}_R} \{\Psi_{j+1}(w)\}
\]

- all other nodes \( u \) keep their current pseudo-time values, i.e., \( \Psi_{j+1}(u) = \Psi_j(u) \).

For example, given \( \psi_0 \) from Figure 8.2(a), if the marked node executes, then the next
pseudo-time values are the ones in Figure 8.2(b).
Observation 8.2.1 The following relations hold:

(i) $\Psi_0(R) \leq h - 1$

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

Corollary 8.2.2 $\Psi_0(v) \leq 2h - 1$, for any node $v$.

Proof. Since in the original tree the number of nodes $n > 1$, the height $h$ is greater than 1, $h \geq 2$. Since for $h > 1$, $h - 1 \leq 2h - 1$, by Observation 8.2.1, and $\Psi_0(R) \leq 2h - 1$. Since $i \leq h$, for any node $v \neq R$, by Observation 8.2.1, $\Psi_0(v) \leq 2h - 1$.

Let $E(v, t)$ be the predicate: “Node $v$ is enabled if $\Psi(v) = t$.”

Observation 8.2.3 (i) If $E(v, t)$ is true then $E(v, t + 2k + 1)$ is false and $E(v, t + 2k)$ is true, for all $k \geq 0$.

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

Given a starting configuration $C_0$ and $C_j$ a configuration after Algorithm SSDST has executed $j$ steps, the relationship between the number of rounds that have elapsed and $\Psi_j$ is given by Property 8.2.4.

Property 8.2.4 Given a starting configuration $C_0$ and $C_j$ some configuration after Algorithm SSDST has executed $j$ steps, then the number of rounds elapsed is $q \leq \min\{\forall \text{ nodes } v, \Psi_j(v)\}$.

Proof. A round has elapsed if all 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.

The $n$ values in the tree (one value per node) to be arranged as a min-heap, can be arranged in a strict sorted order: $r_1 < r_2 < \ldots < r_n$.

We assume that the starting values are distinct. (If necessary, we can add infinitesimal tie-breakers to the values.)

Definition 8.2.2 For any given configuration $C$ of Algorithm A.H.E.AP, let $l_i$ be the level of the node that holds the value $r_i$; we call the function $W(C) = \sum_{i=1}^{n} l_i i$ the weighted path length of the configuration $C$. 

98
Function $W()$ is a strictly positive function. It strictly increases when a swap is executed between some node $v$ that holds the value $r_i$ and some child $w \in D_v$ that holds the value $r_j$, with $r_i < r_j$. The value by which $W$ increases is $j - i$.

By Lemma 7.1, if the heap property is not valid at some node $v$, node $v$ will execute a swap in finite number of rounds.

Since $W(C)$ is an increasing integer function bounded by $hn$, it must converge in finitely many steps. Thus:

**Observation 8.2.5** Function $W$ converges in finitely many of rounds. Let $C_*$ be the configuration after convergence. Then $C_*$ has the heap property.

Let $L_i$ be the level of the node that holds the value $r_i$ in configuration $C_*$. We adapt Definition 4.2.2 of array $pos$ from Chapter 4 to a tree.

**Definition 8.2.3** Given $j, 1 \leq j \leq n$, and some $t \geq 0$, the value $pos[j, t]$ represents the level of node $v$ that currently holds the value $r_j$ and $\Psi(v) = t$.

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

First, we show that once the $\Psi$ value of some node is $t$, the level $pos[j, t]$ of the element $r_j$ is within a certain range (Property 8.2.6). In order to show that Algorithm $A\mathcal{H\mathcal{E}\mathcal{A}}P$ arranges the values as a heap, we show that after $7h/2 - 4$ rounds, $pos[j, t] = L_j$ for all $j$ (Lemma 8.1).

**Property 8.2.6** For any $t \geq 0$ and for any $j, 1 \leq j \leq n$,

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

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

**Proof.** Consider the predicates:

$$
P(t) : \text{for any } j \in 1 \ldots n, pos[j, t] \leq \max \left\{ \frac{L_j}{P[j, t]} \right\}
$$

$$
Q(t) : \text{for any } j \in 1 \ldots n, pos[j, t] \geq \min \left\{ \frac{L_j}{Q[j, t]} \right\}
$$
We first show by induction on $t$ that $P(t)$ holds. Then we show that $Q(t)$ holds.

**Base case:** $0 \leq t \leq 2h - 3$. Since $L_j \leq h$, then $\max(L_j, P[j, t]) = P[j, t] \geq h$. From Definition 8.2.3, $\pos[j, t] \leq h$. Thus $P(t)$ holds for any $t$, $0 \leq t \leq 2h - 3$.

**Inductive step:** $t \geq 2h - 2$ $P(t - 1)$ is true, and we show that $P(t)$ is true.

From the inductive hypothesis, for any $i, 1 \leq i \leq n$ such that $L_i < L_j$, the level of $r_i$ is:

$$\pos[i, t - 1] \leq \max \left\{ \frac{L_i}{P[i, t - 1]} \right\}.$$

Condition $L_i < L_j$ implies that $L_i \leq L_j - 1$, so $P[i, t - 1] \leq P[j, t] - 1$. We can conclude that

$$\max \left\{ \frac{L_i}{P[i, t - 1]} \right\} \leq \max \left\{ \frac{L_j - 1}{P[j, t] - 1} \right\} = M, \text{ where } M = \max \left\{ \frac{L_j}{P[j, t]} - 1 \right\}.$$

Thus all the elements $r_i$ whose levels are smaller than the level of $r_j$ in the configuration $C_*$ are currently situated at levels less or equal to $M$ when their pseudo-times are $t - 1$, i.e. $\forall i \in 1 \ldots n$ such that $L_i < L_j$, $\pos[i, t - 1] \leq M$. Their levels cannot exceed $M$ for $t' \geq t - 1$, since there are no elements whose $L$ values are less than $L_j$ outside the level range $1 \ldots M$. The element $r_j$ at pseudo-time $t - 1$ is situated at the level $\pos[j, t - 1] \leq M + 2$.

By the inductive hypothesis,

$$\pos[j, t - 1] \leq \max \left\{ \frac{L_j}{P[j, t] + 1} \right\} \leq \max \left\{ \frac{L_j + 1}{P[j, t] + 1} \right\} = M + 2.$$

We show now that $\pos[j, t] \leq M + 1$:

$$\pos[j, t] \leq M + 1 = \left\{ \frac{L_j}{P[j, t]} \right\}$$

where $P[j, t] = -t + 2L_j + 3h - 5$.

At pseudo-time $t - 1$, the elements whose $L$ values are smaller than $L_j$ are at levels less than $M + 1$, thus the $L$ values of the elements at level $M + 1$ and $M + 2$ are at least $L_j$.

We have two cases:

1) $\pos[j, t - 1] = M + 2$. Thus there exists an element $r_k \neq r_j$ situated at level $M + 1$ ($\pos[k, t - 1] = M + 1$), held by a node $u$ that is the parent of the node $v$ that holds the element $r_j$ situated at level $M + 2$. Since the element $r_k$ has $L_k > L_j$, and the tree has the min-heap property in configuration $C_*$, it follows that $r_k > r_j$.
Since the elements smaller than \( r_j \) would have their \( L \) values smaller than \( L_j \), it implies that at pseudo-time \( t - 1 \) node \( u \) does not have any children with values smaller than \( r_j \). Thus element \( r_j \) is the smallest among all the children of node \( u \). If \( E(u, t - 1) \) holds then node \( u \) will execute and swap the values \( r_k \) and \( r_j \), thus \( pos[j, t] = M + 1 \). If \( E(u, t - 1) \) is false, since \( t \geq 2h - 2 \) (so \( t - 1 \geq 2h - 3 \) and \( 2h - 3 \geq \Psi(v) \) for all nodes \( v \)), by Observation 8.2.3 then \( E(u, t - 2) \) holds. Node \( u \) has executed at pseudo-time \( t - 2 \) and as a result, \( pos[j, t - 1] = M + 2 \). Thus the value \( r_k \) at level \( M + 1 \) at pseudo-time \( t - 1 \) is smaller than \( r_j \) and \( L_k \leq L_j \). Contradiction.

2) \( pos[j, t - 1] = M + 1 \). Since the elements situated at levels \( M + 2 \) and higher are smaller than \( r_j \), the element \( r_j \) cannot move to \( M + 2 \) in any of the subsequent times.

In both cases, \( pos[j, t] \leq M + 1 \), thus Predicate \( P(t) \) holds.

The proof that \( Q(t) \) holds is similar.

**Base case:** \( 0 \leq t \leq 2h - 3 \). Since \( L_j \geq 1 \), then \( min(L_j, Q[j, t]) = Q[j, t] \leq 0 \). From Definition 8.2.3, \( pos[j, t] \geq 1 \). Thus \( P(t) \) holds for any \( t, 0 \leq t \leq 2h - 3 \).

**Inductive step:** \( t \geq 2h - 2 \). Predicate \( Q(t - 1) \) is true, and we show that \( Q(t) \) is true.

From the inductive hypothesis, for any \( i, 1 \leq i \leq n \) such that \( L_i > L_j \), the level of \( r_i \) is:

\[
pos[i, t - 1] \geq min \left\{ \frac{L_i}{Q[i, t - 1]} \right\}.
\]

Condition \( L_i > L_j \) implies that \( L_i \geq L_j + 1 \), so \( Q[i, t - 1] \geq Q[j, t] + 1 \). We can conclude that

\[
min \left\{ \frac{L_i}{Q[i, t - 1]} \right\} \geq min \left\{ \frac{L_j + 1}{Q[j, t] + 1} \right\} = M, \text{ where } M = min \left\{ \frac{L_j}{Q[j, t]} + 1 \right\}.
\]

Thus all the elements \( r_i \) whose levels are larger than the level of \( r_j \) in the configuration \( C_0 \) are currently situated at levels greater or equal to \( M \) when their pseudo-times are \( t - 1 \), i.e.\( \forall i \in 1...n \) such that \( L_i > L_j \), \( pos[i, t - 1] \geq M \).

Their levels cannot decrease below \( M \) for \( t' \geq t - 1 \), since there are no elements whose \( L \) values are larger than \( L_j \) outside the level range \( M \ldots h \). The element \( r_j \) at pseudo-time \( t - 1 \) is situated at the level \( pos[j, t - 1] \geq M - 2 \).

By the inductive hypothesis,

\[
pos[j, t - 1] \geq min \left\{ \frac{L_j}{Q[j, t] - 1} \right\} \geq min \left\{ \frac{L_j - 1}{Q[j, t] - 1} \right\} = M - 2.
\]
We show now that \( \text{pos}[j, t] \geq M - 1 \):

\[
\text{pos}[j, t] \geq M - 1 = \begin{cases} L_j \\ P[j, t] \end{cases}
\]

where \( Q[j, t] = t + 2L_j + 3 - 4h \).

At pseudo-time \( t - 1 \), the elements whose \( L \) values are larger than \( L_j \) are at levels larger than \( M - 1 \), thus the \( L \) values of the elements at level \( M - 1 \) and \( M - 2 \) are at most \( L_j \). We have two cases:

1) \( \text{pos}[j, t - 1] = M - 2 \). Let \( v \) be the node situated at level \( M - 2 \) that holds the element \( r_j \). Thus the children of node \( v \) situated at level \( M - 1 \) hold only values \( r_k \) such that \( L_k \leq L_j \). Since the tree has the min-heap property in configuration \( C_\ast \), it follows that all the children of node \( v \) hold values less than \( r_j \). Let \( w \) be the child of node \( v \) that holds the element \( r_k \) that is the smallest value among the children of node \( v \), and \( \text{pos}[k, t - 1] = M - 1 \).

If \( E(v, t - 1) \) holds then node \( v \) will execute and swap the values \( r_k \) and \( r_j \), thus \( \text{pos}[j, t] = M - 1 \). If \( E(v, t - 1) \) is \( \text{false} \), since \( t \geq 2h - 2 \) (so \( t - 1 \geq 2h - 3 \) and \( 2h - 3 \geq \Psi(v) \) for all nodes \( v \)), by Observation 8.2.3 then \( E(v, t - 2) \) holds. Node \( v \) has executed at pseudo-time \( t - 2 \) and as a result, \( \text{pos}[j, t - 1] = M - 2 \). Thus the value \( r_k \) at level \( M - 1 \) at pseudo-time \( t - 1 \) is larger than \( r_j \) and \( L_k \geq L_j \). Contradiction.

2) \( \text{pos}[j, t - 1] = M - 1 \). Since the elements situated at levels \( M - 2 \) and lower are larger than \( r_j \), the element \( r_j \) cannot move down to \( M - 2 \) in any subsequent steps.

In both cases, \( \text{pos}[j, t] \geq M - 1 \), thus Predicate \( Q(t) \) holds. \( \square \)

**Lemma 8.1** Algorithms \( \mathcal{AHEAP} \) arranges the elements as a min-heap in \( 7h/2 - 4 \) rounds; thus the stabilization time is \( O(h) \) rounds.

**Proof.** By Definition 8.2.3, \( 1 \leq \text{pos}[j, t] \leq h \), \( \forall t \geq 0 \), and \( \forall j \in 1...n \). From Property 8.2.6, for \( t = 7h/2 - 4 \) and \( L_j \leq h/2 + 1 \), thus \( \text{pos}[L_j, 7h/2 - 4] \leq L_j \). For \( L_j = 1 \), \( \text{pos}[j, 7h/2 - 4] \leq 1 \), thus \( \text{pos}[j, 7h/2 - 4] = 1 = L_j \). It follows that for all \( L_j \leq h/2 + 1 \), \( \text{pos}[j, 7h/2 - 4] = L_j \).
From Property 8.2.6, for $t = 7h/2 - 4$ and $L_j \geq h/2 + 1$, thus $pos[j, 7h/2 - 4] \geq L_j$. For $L_j = h$, $pos[j, 7h/2 - 4] \geq h$, thus $pos[j, 7h/2 - 4] = h = L_j$. It follows that for all $L_j \geq h/2 + 1$, $pos[j, 7h/2 - 4] = L_j$. Thus for all $j \in 1 \ldots n$, $pos[j, 7h/2 - 4] = L_j$.

The value $7h/2 - 4$ is the value of the pseudo-time function for some node $v$. By Property 8.2.4, after at most $7h/2 - 4$ rounds, the values are arranged as a min-heap.

8.3 Reduction of Algorithm \texttt{HEAP} to \texttt{A-HEAP}

In this section we first show that Algorithm \texttt{HEAP} reduces to Algorithm \texttt{A-HEAP}. We can then conclude that, starting from an arbitrary configuration, in at most $14h - 16$ rounds, Algorithm \texttt{HEAP} arranges the values as a min-heap (Lemma 8.4).

We use Definition 2.1.1 of reduction from Chapter 2.

If \texttt{A'} accomplishes a task in the model \texttt{M'} and \texttt{A} reduces to \texttt{A'}, then by Definition 2.1.1, \texttt{A} accomplishes the same task in the model \texttt{M}.

We now show that Algorithm \texttt{HEAP} reduces to Algorithm \texttt{A-HEAP}.

Let $S_v = (s_v, x_v, c_h_v)$ be the set of all variables of node $v$ in order ($S, IV, child$) used by Algorithm \texttt{A-HEAP} in the abstract model. Let $S^v = (s_v, x_v, t_v, c_h_v)$ be the set of all variables of node $v$ in order ($S, IV, tmpS, child$) used by Algorithm \texttt{HEAP} in the shared-memory model.

Then $R$ is defined as follows. $R(S_1, \ldots S_n) = \{(S^1_1, \ldots S^1_n), t_i \in \{A, B\}, 1 \leq i \leq n\}$.

For each configuration $C_1$ of Algorithm \texttt{A-HEAP} in the abstract model, there exists $2^n$ configurations in $R(C_1)$ of Algorithm \texttt{HEAP} in the shared-memory model, thus Condition (i) of Definition 2.1.1 is satisfied.

We are left to show that Condition (ii) of Definition 2.1.1 is satisfied (Lemma 8.2).

**Lemma 8.2** Given $C_1$ and $C_2$, two configurations of Algorithm \texttt{A-HEAP} in the abstract model, such that $C_1 \rightarrow C_2$ is an execution step of Algorithm \texttt{A-HEAP}; for any configuration $C'_1 \in R(C_1)$, if Algorithm \texttt{HEAP} 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.** The proof is similar to the proof of Lemma 4.2 in Chapter 4.

103
A node state contains all the variables stored at that node. The system configuration contains the states of all the nodes. An execution step is a transition from one configuration to another. We break the system configuration into a number of chunks. A chunk is a set of a node and its descendants in the tree such that:

- the first node in each chunk is enabled, and
- all the descendants of the first node reachable by a path of disabled nodes are added to the chunk.

We build the set of chunks starting from the root in depth-first-search (DFS) order. If the root node is currently disabled, then the root and all nodes reachable from the root reachable by a path of disabled nodes are not part of any chunk. We call the set of those nodes the null chunk.

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

From Property 7.1.2 we know that if a non-leaf node is enabled, then its children are disabled. So except for the leaf nodes, every chunk contains at least two nodes. If the chunk contains at least two nodes, then the last node in the chunk is disabled, so it cannot affect the state of the first node of other chunks.

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

Let $C_1 = (H_1^1, H_2^1, \ldots, H_k^1)$ be the set of chunks, omitting the null chunk. Let $v$ the first node in some chunk $H_i^1$ of Configuration $C_1$ of Algorithm \textit{ASORT} in the abstract model. Assume without loss of generality that Action $BAA$ is enabled at $v$.

- If $H_i^1$ contains a single node (node $v$ is a leaf node), then even if the node $v$ is enabled, $v$ will just change its $S$ (from $A$ to $B$) without the swap (since it has no children).

The execution step of Algorithm \textit{ASORT} is: $((A, x_v, \bot)) \xrightarrow{\text{BAA}(v)} ((B, x_v, \bot))$. In the shared-memory model this corresponds to: $((A, x_v, \rightarrow, \bot)) \xrightarrow{\text{BAA}(v)} ((B, x_v, \rightarrow, \bot))$ that starts in any configuration of $R(C_1)$ restricted to the chunk $H_i^1$, and ends in some configuration of $R(C_2)$ restricted to the chunk $H_i^1$.
If the chunk contains more than one node: $H^1_v = (S_v, S_1, S_2, \ldots S_n_v)$ where $S_1$ is the state of a node $w$ that is a child of node $v$ that holds the minimal $IV$ value among all the children of node $v$. (Since node $v$ is a non-leaf node, all its children are disabled, and part of node $v$'s chunk.) For the ease of notation, assume $S_v = (A, x, c_1)$ and $S_1 = (A, y, c_2)$.

We have two cases, depending on whether $v$ and $w$ violate the heap property.

1) nodes $v$ and $w$ do not violate the heap property, i.e., $IV.w \geq IV.v$.

Then the execution step of Algorithm $A.SORT$ is: $((A, x, c_1), (s_1, y, c_2)) \xrightarrow{BAA(v)} ((B, x, c_1), (s_1, y, c_2))$. In the shared-memory model this corresponds to:

$((A, x, c_1), (A, y, c_2)) \xrightarrow{BAA(v)} ((B, x, c_1), (A, y, c_2))$ that starts in any configuration of $R(C_1)$ restricted to the chunk $H^1_v$, and ends in some configuration of $R(C_2)$ restricted to the chunk $H^1_v$.

2) nodes $v$ and $w$ violate the heap property, i.e., $IV.w < IV.v$. Then the execution step of Algorithm $A.SORT$ is: $((A, x, c_1), (A, y, c_2), S_2, \ldots S_n_v) \xrightarrow{BAA(v)} ((B, y, c_1), (A, x, c_2), S_2, \ldots S_n_v)$. ($v$ can affect only the variables of node $w$).

In the shared-memory model this corresponds to:

$((A, x, c_1), (A, y, c_2), S_2, \ldots S_n_v) \xrightarrow{BAA(v)}$ \hspace{2cm} $((X, x+y, B, c_1), (A, y, c_2), S_2, \ldots S_n_v) \xrightarrow{S1(w)}$ \hspace{2cm} $((X, x+y, B, c_1), (Y, x, A, c_2), S_2, \ldots S_n_v) \xrightarrow{S2(v)}$ \hspace{2cm} $((B, y, c_1), (Y, x, A, c_2), S_2, \ldots S_n_v) \xrightarrow{S3(w)}$ \hspace{2cm} $((B, y, c_1), (A, x, c_2), S_2, \ldots S_n_v)$

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

\[ \square \]

If the starting state of the node is either $A$ or $B$, then the value to be sorted is its initial value. If some node starting state is either $X$ or $Y$, then it is possible for some of the three steps of the swap to be applied (see Section 8.1) 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.

Some properties of Algorithm $SORT_e$ from Chapter 4 implemented in the shared-memory model are valid for Algorithm $\mathcal{HEAP}$ as well.

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

Property 8.3.1 shows that for any node $v$ whose $S$ is $X$, either $S$ remains $X$ and then the node $J.v$ will have its $S$ equal to $Y$ in at most three rounds (by executing Action $S1$), or $v$ changes its $S$ to $A$ or $B$ in at most one round.

Property 8.3.2 shows that for any node $v$ such that $S.v = X$ and $S(J.v) = Y$ then $IV.v$ gets the value $IV(J.v)$ and then $v$ changes its $S$ to $A$ or $B$ in at most one round. Node $J.v$ had already stored in $IV(J.v)$ the old value of $IV.v$ (by executing Action $S1$) and by Property 8.3.3 will restore its $S$ from $Y$ to either $A$ or $B$ (depending on the value of $\text{tmpS}$) in at most one round. We can then conclude that if $S.v$ is either $X$ or $Y$, then in at most four rounds $S.v$ is either $A$ or $B$ (Lemma 8.3).

**Property 8.3.1** For any node $v$ where $S.v = X$, either $S$ remains $X$ and then variable $S$ of node $J.v$ node becomes $Y$ in at most three rounds (by executing Action $S1$ at node $J.v$), or $v$ changes its $S$ to $A$ or $B$ in at most one round.

**Proof.**

Let $(s_0, x, t_0, c_0)$, respectively $(s_1, y, t_1, c_1)$, be the set of variables of node $v$, respectively $J.v$ (if $J.v \neq \bot$) in the order $(S, IV, \text{tmpS}, \text{child})$. Since the values of variable $\text{tmpS}$ are drawn from the set $\{A, B\}$, we have that $t_0, t_1 \in \{A, B\}$.

We analyze, by cases, what happens to the pair $(v, J.v)$ when Algorithm $\mathcal{HEAP}$ is executed locally. An execution step is marked by an arrow $\rightarrow$ labeled by a action that is enabled and has been selected for execution by the respective node. If more than one action is enabled at some node $v$, then we use split arrows to show possible execution steps. The underscore symbol $\_\_$ means that the respective value does not matter.

We have the following cases:
1. \( J.v = \bot \). Then the only enabled action at node \( v \) is Action \( C2 \): \[ ((X, -, t_0, -), \bot) \xrightarrow{C2(v)} ((t_0, -, -, -), \bot). \]

2. \( J.v \neq \bot \land s_1 = Y \). Done.

3. \( J.v \neq \bot \land s_1 \in \{A, B\} \). Then node \( v \) is currently disabled and only Action \( S1 \) is enabled at node \( J.v \):
\[ ((X, -, -, -), (s_1, -, -, -)) \xrightarrow{S1(J.v)} ((X, -, -, -), (Y, -, s_1, -)). \] Done.

4. \( J.v \neq \bot \land s_1 = X \). Then only Action \( C3 \) is enabled at \( v \). Node \( J.v \) may have one of the following actions enabled: Action \( S2 \), Action \( C2 \) or Action \( C3 \). But the execution of these actions does not affect the variables of node \( v \). Also, since one cannot know what the values of the variables of node \( J.v \) will be after Action \( C3 \) is executed at node \( v \), we use the underscore symbol.
\[ ((X, -, t_0, -), (X, -, -, -)) \xrightarrow{C3(v)} ((t_0, -, t_0, -), (-, -, -)). \]

\( \Box \)

**Property 8.3.2** For any node \( v \) where \( S.v = X \), Action \( S2 \) is enabled at \( v \) if and only if \( v \) has a child \( J.v \) whose \( S \) has value \( Y \). Once Action \( S2 \) is executed at node \( v \), \( S.v \) becomes either \( A \) or \( B \).

**Proof.** The property follows from the Action \( S2 \) of Algorithm \( \text{HEAP} \). If \( S.v = X \land J.v \neq \bot \land S(J.v) = Y \), then the only enabled action at \( v \) is Action \( S2 \), and node \( J.v \) is currently disabled. Once Action \( S2 \) executes in at most one round, \( S.v \) becomes either \( A \) or \( B \), and we are done. \( \Box \)

**Property 8.3.3** If \( S.v = Y \), then in at most two rounds, \( S.v \) becomes either \( A \) or \( B \).

**Proof.** Let \((s_0, z, t_0, c_0)\), respectively \((s_2, z, t_2, c_2)\), be the set of variables of node \( v \), respectively its parent \( p_v \) (if \( p_v \neq \bot \)). Again, \( t_0, t_2 \in \{A, B\} \).

We analyze, by cases, what happens to the pair \((p_v, v)\) when Algorithm \( \text{HEAP} \) is executed. We have the following cases:
1. \( p_v = \bot \). Then \( (\bot, (Y, \rightarrow, t_0, c_0)) \xrightarrow{c_{1(\nu)}} (\bot, (t_0, \rightarrow, c_0)) \), and we are done.

2. \( p_v \neq \bot \land s_2 \in \{A, B\} \). Then Action S3 is the only enabled action at node \( v \). Node \( p_v \)
may have Action S1 enabled while Action S3 is executed at node \( v \). But since an execution of node \( p_v \)
does not affect the variables of node \( v \), and since one cannot know what the values of the variables of node \( p_v \)
will be after Action S3 is executed at node \( v \), we use the symbol underscore:
\[
((s_2, \rightarrow, t_2, c_2), (Y, \rightarrow, t_0, c_0)) \xrightarrow{s_{3(\nu)}} ((\_, \rightarrow, c_2), (t_0, \rightarrow, c_0))
\]
and we are done.

3. \( p_v \neq \bot \land s_2 = X \). Node \( v \) is disabled until node \( p_v \) executes Action S2. Then by
Property 8.3.3, node \( p_v \) executes Action S2 and the value of \( S.(p_v) \) change to either A or B. We then apply case 2.

4. \( p_v \neq \bot \land s_2 = Y \). Only Action S3 is currently enabled at node \( v \). Node \( p_v \) may have
Action S3 or Action C1 enabled while Action S3 will be executed at node \( v \). But since an execution of node \( p_v \)
does not affect the variables of node \( v \), and since one cannot know what the values of the variables of node \( p_v \)
will be after Action S3 is executed at node \( v \), we use the underscore symbol:
\[
((Y, \rightarrow, c_2), (Y, \rightarrow, t_0, c_0)) \xrightarrow{s_{3(\nu)}} ((\_, \rightarrow, c_2), (t_0, \rightarrow, c_0))
\]
and we are done.

\[\Box\]

**Lemma 8.3** For any node \( v \), if \( S.v \in \{X, Y\} \), in at most four rounds \( S.v \) becomes either A or B.

**Proof.** Follows from Properties 8.3.1, 8.3.2, and 8.3.3. \[\Box\]

**Lemma 8.4** Starting from an arbitrary configuration, in at most \( 14n - 16 \) rounds, Algorithm \( \text{HEAP} \) arranges the \( n \) values in min-heap order.

**Proof.** From Lemma 8.3, each swap takes at most 4 rounds. From Lemma 8.1, if a swap takes at most 1 round, then heap takes at most \( 7n/2 - 4 \) rounds. Since the swap takes at most 4 rounds, we obtain a total of at most \( 14n - 16 \) rounds. \[\Box\]
8.4 Snap-Stabilizing Max-Heap Algorithm

In this section we present a snap-stabilizing max-heap algorithm in a rooted binary tree (Algorithm Heap). The maximum number of items that can be stored at any time at any node is independent of the size $n$ of the network.

8.4.1 Variables, Constants, and Configurations

In the binary tree model considered, each node $i$ holds four constants that are not changed by the BST algorithm. We assume that they cannot be corrupted by a transient fault either.

The constants are: the value $V.i$ that needs to be sorted in the tree, the parent ID $p.i$, the left child ID $left.i$, and the right child ID $right.i$. If $i$ does not have any of the above three neighbors, the corresponding constant's value is represented as $\bot$. For example, for the root node $r$, $p.r = \bot$, and for the leaf nodes, $left.i = right.i = \bot$.

We denote the set of neighbors and set of children of $i$ by $N.i$ and $D.i$, respectively. We assume that the tree has $n$ nodes and has a height of $h$. Let $T_i$ be the subtree rooted at $i$. Then $s_i$ and $h_i$ denote the number of nodes and height, respectively, of $T_i$.

The variables used by some node $i$ are described below. The heap value $HV.i$ is the result of the heapify process (Algorithm Heap). Variable tmp is used for exchanging data with the parent or the children.

All the variables require $O(1)$ space. Algorithm BST needs to maintain the size of the subtrees rooted at each node. This size variable $s.i$ for node $i$ is computed in Algorithm Heap and used in Algorithm BST. The Heap construction does not use this variable but for the sake of algorithm simplicity we add the calculation of this variable to it.

Algorithm Heap uses six states (see Figure 8.3): $C$ (cleaning state), $B$ (ready to start the heapify process), $M$, $M^{left}$, $M^{right}$ (the states corresponding to if the maximum heap value $HV$ is based on its own heap value, the maximum heap value of its left child, the maximum heap value of its right child, respectively), $P$ (the Heap phase finished at this node, and the Sort phase is ready to start at this node).
A configuration in which the root is in state $C$ is called a \textit{clean configuration}. Starting from such a configuration, all other nodes in the tree will eventually reach $C$ state. If all nodes are in $C$ state, then the corresponding configuration is termed as a \textit{normal starting configuration}. Any configuration reachable from a normal starting configuration by executing the algorithm guards is called a \textit{normal configuration}. All other configurations are considered to be \textit{abnormal}.

Some abnormal configurations can be locally detected by the processors. This local detection is implemented using the \textit{abnormal} predicates in Algorithm \textit{Heap}. These predicates are used as guards of correction actions in order to avoid possible deadlocks and to speed up the protocol. Unfortunately, some problems of abnormal configurations cannot be locally detected. For example, the initial configuration may contain some sorted values (in tSV) that do not match any $V$ values. The correction actions can remove the locally detectable problems in $O(h)$ rounds even before the root executes its initialization action. The other problems are eventually removed during the suffix of the protocol starting from the initialization action of the root.

Starting from an abnormal configuration, an execution not necessarily will bring the system to a normal starting configuration, but to a normal configuration. When a node has an abnormal predicate enabled, it will change its state to $C$, and all the nodes in its subtree will enter $C$ state, but not necessarily its parent (e.g. if the parent state is $B$).

\subsection*{8.4.2 Algorithm Heap}

We sketch the steps of a normal execution of module \textit{Heap} starting from a clean configuration. Upon receiving an external command to sort, if the root is enabled to start the BST protocol, it starts the heapify process (module \textit{Heap}). The root is enabled to initiate if it is in $C$ and its children are in $C$. The root broadcasts the heapify command by changing
its state to $B$. As this message (wave) goes down the tree, all internal nodes change their state from $C$ to $B$. When this broadcast wave reaches the leaf nodes, they change their state from $C$ to $M$ to initiate the heapify process (or wave). During this upward wave, the nodes compute two things: the heap value (the maximum value in their subtrees) and the size of their subtrees. When this wave reaches the root, the root changes its state to $M$ and the heap is created. The root then initiates another top-down wave by changing its state from $M$ to $P$. The next phase, i.e., the BST construction phase starts from the $P$ state.

We now describe the heap construction in more detail by referring Algorithm 8.4.1.

**Algorithm 8.4.1 Module Heap**

**Predicates**

- **abnormal.B**($i$) :: is true when the node, in state $B$, is in abnormal situation with some neighbor (parent or child).

  \[
  \text{abnormal.B}(i) \equiv (p.i \neq \perp \land S.p.i \neq B) \lor (D.i \neq \perp \land \exists j \in D.i : S.j \notin \{C, B, M, M^\text{left}, M^\text{right}\}) \lor HV.i \neq V.i
  \]

- **abnormal.M**($i$) :: is true when the node, in state $M$, either has some variables with abnormal values or is in abnormal situation with some neighbor (parent or child).

  \[
  \text{abnormal.M}(i) \equiv ab.M^*(i) \lor (D.i = \perp \land ab.M^* \text{left}(i)) \lor (D.i = \perp \land ab.M^* \text{nonleft}(i)) \lor (p.i = \perp \land ab.M^* \text{nonright}(i))
  \]

- **ab.M**($i$) :: is true when $HV.i = \perp \lor \text{left} \land \text{left} \lor \text{right} \land \text{right} \lor s.i < 0$.

  \[
  \text{ab.M}(i) \equiv \text{HV.i} = \perp \lor (\text{left} \land \text{left} \lor \text{right} \land \text{right} \lor s.i < 0)
  \]

- **ab.M**.$\text{left}$($i$) :: is true when $s.i > 1$.

- **ab.M**.$\text{nonleft}$($i$) :: is true when $s.i = \perp \lor \text{down} \lor \text{left} \lor \text{left} \lor \text{right} \lor \text{right}$.

  \[
  \text{ab.M}^* \text{.nonleft}(i) \equiv s.p.i \notin \{B, M, M^\text{left}, M^\text{right}, P\}
  \]

1. *(Start building a Heap)* If the root is in $C$, its children will change to $C$ in at most one round. Either Action $aCm$ or $aCb$ is enabled, and since it is the only enabled action, it is eventually executed in at most one round. When its children change to $C$, the root changes its state from $C$ to $B$ and sets $HV$ to its internal value $V$ (Action $CB$). An internal node changes its state from $C$ to $B$ when its parent is in $B$ and its children are in $C$. It also

111

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
consistency(i) :: is true when the nodes stores in $rHV.i$ and $lHV.i$ the heap values of its children (if any).

$$\text{consistency}(i) \equiv (\text{left}.i \neq \bot \land lHV.i = HV.left.i) \land (\text{right}.i \neq \bot \land rHV.i = HV.right.i)$$

h.order(i) :: is true when the node has the MaxHeap property.

$$\text{h.order}(i) \equiv HV.i \neq \bot \land lHV.i \leq HV.i \land rHV.i \leq HV.i$$

update.HVs(i) :: is true when the node needs to update its heap value since some child has a bigger heap value than itself.

$$\text{update.HVs}(i) \equiv (HV.i < lHV.i \lor HV.i < rHV.i) \land ((d = \text{left} \land S.p.i \in \{M, M^{\text{right}}\}) \lor (d = \text{right} \land S.p.i \in \{M, M^{\text{left}}\}))$$

Macros

init(i) :: is executed when changing from $B/C$ to $M$ state.

1. $down.i = \bot$
2. $s.i = 1$
3. if $\text{left}.i \neq \bot$ then $s.i+ = s.left.i$
4. if $\text{right}.i \neq \bot$ then $s.i+ = s.right.i$
5. if ($\text{left}.i = \bot$) then $lHV.i = \bot$ else $lHV.i = HV.left.i$
6. if ($\text{right}.i = \bot$) then $rHV.i = \bot$ else $rHV.i = HV.right.i$

$s.i$ is computed based on the children’s variables $s.left.i$ and $s.right.i$, and variables $lHV$ and $rHV$ are set to $HV.left.i$ respectively $HV.right.i$ for BST construction. Any non-leaf node sets $\text{dir}$ to $\text{right}$ since the sorted values sent in decreasing order by the root will fill the nodes in right-parent-left order.

init.P(i) :: is executed when changing from $M$ to $P$ state to prepare the node for BST construction.

1. $down.i = \bot$
2. if $D.i \neq \bot$ then $\text{dir}.i = \text{right}$ else $\text{dir}.i = \bot$
3. $tSV.i = \bot$

set.HVs(i) :: selects the child $\text{dir} \in \{\text{left}, \text{right}\}$ that has the maximum heap value by comparing $lHV.i$ and $rHV.i$.

1. $\text{dir}.i = \text{MAX}(i)$
2. if $(\text{dir}.i = \text{left} \land \text{left}.i \neq \bot \land HV.i < lHV.i)$ then
3. // the maximum heap value is in the left subtree so change $HV.i$
4. // to that value and select in $lHV.i$ the maximum between the
5. // old $HV.i$ and the maximum value left in the left subtree
6. $\text{down}.i = HV.i$
7. $HV.i = lHV.i$
8. $\text{tmp}.i = \text{MAX}(\text{left}.i)$
9. if $\text{tmp}.i = \text{left}$ then
10. if $\text{down}.i < lHV.left.i$ then $lHV.i = lHV.left.i$ else $lHV.i = \text{down}.i$
11. if $\text{tmp}.i = \text{right}$ then
12. if $\text{down}.i < rHV.left.i$ then $rHV.i = rHV.left.i$ else $rHV.i = \text{down}.i$
13. $S.i = M^{\text{left}}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if (dir.i = right ∧ right.i ≠ ∩HV.i < rHV.i) then

// the maximum value is in the right subtree so change HV.i to that value and
// select in rHV.i the maximum between the old HV.i value and the maximum value
// left in the right subtree

down.i = HV.i
HV.i = rHV.i

if tmp = left then
  if down.i < lHV.right.i then rHV.i = lHV.right.i else rHV.i = down.i
if tmp = right then
  if down.i < rHV.right.i then rHV.i = rHV.right.i else rHV.i = down.i

S.i = M.right

\{Program for the root node r\}

\(CB: \quad S.r = C ∨ \forall j ∈ D, S.j = C \quad \rightarrow \quad S.r = B; HV.r = V.r\)

\(BM*: \quad \neg abnormal.B(r) ∧ S.r = B ∨ \forall j ∈ D, S.j = \{M, M.left, M.right\} \quad \rightarrow \)
\(\quad \text{init}(r), \quad S.r = M, \quad \text{set.HV}s(r)\)

\(M^{tr}M: \quad \neg abnormal.M^*(r) ∧ ∃ j ∈ \{left, right\} (S.r = M^j ∧ j.r ≠ ∩ ∧ HV.j.r = down.r) \quad \rightarrow \)
\(\quad S.r = M\)

\(MP: \quad \neg abnormal.M^*(r) ∧ S.r = M ∧ \forall j ∈ D, S.j = M ∧ \text{consistency}(r) \quad \rightarrow \)
\(\quad S.r = P, \quad \text{init.P}(r)\)

\(aCm: \quad (S.r = B ∧ abnormal.B(r)) ∨ (S.r ∈ \{M, M.left, M.right\} ∧ abnormal.M^*(r)) \quad \rightarrow \)
\(\quad S.r = C\)

\{Program for an internal node i, which is the d child of its parent,\n d ∈ \{left, right\}\}

\(CB: \quad S.i = C ∧ S.p.i = B ∧ \forall j ∈ D, S.j = C \quad \rightarrow \quad S.i = B; HV.i = V.i\)

\(BM*: \quad \neg abnormal.B(i) ∧ S.i = B ∧ S.p.i = B ∧ \forall j ∈ D, S.j = \{M, M.left, M.right\} \quad \rightarrow \)
\(\quad \text{init}(i); S.i = M; \text{set.HV}s(i)\)

initializes its heap value HV with its input (or initial) value V (Action CB).

Figure 8.4(a) shows the clean configuration for a 11-node binary tree. After B wave is
$M^r M \quad -\text{abnormal} \_ M^* (i) \land \exists j \in \{\text{left}, \text{right}\} (S_i = M^j \land j \not= i \land HV_j = \text{down}) \rightarrow S_i = M$

$lr M \quad -\text{abnormal} \_ M^* (i) \land S_i = M \land S.p.i = M^d \land h.order (i) \rightarrow HV_i = \text{down}.p.i$

$MM^* \quad -\text{abnormal} \_ M^* (i) \land S_i = M \land \text{update} \_ HV s (i) \rightarrow \text{set} \_ HV s (i)$

$MP \quad -\text{abnormal} \_ M^* (i) \land S_i = M \land S.p.i = P \land \forall j \in D_i S_j = M \land \text{consistency} (i) \rightarrow S_i = P; \text{init} \_ P (i)$

$aCm \quad (S_i = B \land \text{abnormal} \_ B (i)) \lor (S_i \in \{M, M^{\text{left}}, M^{\text{right}}\} \land \text{abnormal} \_ M^* (i)) \rightarrow S_i = C$

executed top-down, the tree state is shown in Figure 8.4(b). We show only the node’s internal value $V$, state $S$, and heap value $HV$. Symbol $^*$ means that the value is not important.

![Figure 8.4: Initial stage of constructing the heap.](image)

(a) Clean configuration. (b) $B$ wave is executed top-down.

2. (Calculating heap and $s.i$ values) A leaf node $i$ changes its state from $C$ to $M$ and executes macro $\text{init}(i)$ (Action $CM$). In the macro $\text{init}(i)$, the node $i$ sets the size of its
subtree, s.i to 1 and sets the heap values of its left (IHV) and right (rHV) subtrees to ⊥ (indicating a non-existent value).

When a parent of a leaf node detects that all its children are in state M (Action BM* is enabled), it executes macro init(i), change from B to M, and executes macro set_HVs(i). If the (parent) node holds a value smaller than any of the heap values of its children, it chooses as its heap value the larger heap value (IHV or rHV) among its children and pushes its own heap value (HV) toward the child that was holding the larger heap value. This heapification process goes up the tree until it reaches the root.

Following Figure 8.4(b), if we assume that all leaves move in the same time, in Figure 8.5(a) we show how the M wave starts from the leaf nodes. For each node we show the variable s, the state S, IHV, HV, rHV, and down. Symbol b means ⊥. Following one more step, in Figure 8.5(b) the parents of the leaf nodes collect the maximum heap values and change their states from B to M/MLef/Mright.

Predicate update_HVs(i) is true when due to the heapification process at the parent of i, i's heap value became smaller than the values of its children. So, HV.i needs to be
swapped with that of one of its children. Predicate $horder(i)$ is true if $i$ satisfies the heap property with respect to its children.

For a non-leaf node $i$ that is about to execute the macro $set.HVs(i)$, we consider three cases.

Case 1) $HV.i$ is larger than the heap values of its children. So, heap order is maintained at $i$. Then the macro $set.HVs(i)$ does not change the variables $S.i$ (remains $M$) and $HV.i$.

For example, consider in Figure 8.6 the following snapshot of some portion of the 11-node tree from Figure 8.4(a). The checkmark symbol marks an enabled node. If the node state is $B$, the values drawn are $V$, $S$ and $HV$. If the node state is $M$, the values drawn are $s$, $S$, $lHV$, $HV$ and $rHV$. The node whose $V = 100$ changes its state from $B$ to $M$ and since the heap values of its children are smaller, the node is done.

Case 2) Assume that the heap value of one of the children (say, the right child $right.i$) of $i$ is higher than both $HV.i$ and that of the left child of $i$. The macro $set.HVs(i)$ selects $dir.i = right$ and sets $S.i = M^{right}$. So, node $i$ will push its old heap value (now in variable $down.i$) to its right child. Assume that $down.i$ is larger than the heap values of the children of $right.i$. So, $down.i$ (the old value of $i$) needs to be pushed only one level down the tree where it becomes the new heap value of $right.i$ in at most two rounds: First Action $lrM$ is performed at $right.i$, then Action $M^{right}M$ is executed at node $i$ ($i$ changes its state back to $M$). Figure 8.7 shows a part of a binary tree to illustrate this case. For each node we show the variable $s$, the state $S, lHV, HV, rHV$, and $down$. Symbol $b$ means $\bot$. The checkmark symbol marks an enabled node.

![Figure 8.6: Macro set.HVs executed at node with $V = 100$](image.png)
Figure 8.7: Macro set_HVs executed at the node with $V = 145$

**Case 3.** Similar to Case 2 except that $down_i$ is smaller than the heap value of one of the children of $right_i$. So, the old value of $i$ (now in $down_i$) needs to be pushed at least two levels down the tree before it finds a node $j$ where $down_i$ becomes the heap value of $j$. In Figure 8.8, the value 130 is pushed down two levels. For each node we show the variable $s$, the state $S$, IHV, HV, rHV, and $down$. Symbol $b$ means $\bot$. The checkmark symbol marks an enabled node.

Figure 8.8: Macro set_HVs executed at node with $V = 130$

Smaller values may be pushed to a node $i$ from its ancestor. When that happens, $i$ changes its state from $M$ to $M_{left}/M_{right}$. When the wave (changing state from $B$ to $M$) reaches the root, the root changes its state from $B$ to $M$. Then the root may change to state $M_{left}$ or $M_{right}$ if it needs to push its heap value (which is its internal value and now in $down_r$) down the tree. Then it pushes $down_r$ to either $M_{left}$ or $M_{right}$. When the corresponding child of the root receives the value $down$, the root goes back to $M$ and stays in $M$ since it has no ancestors.

### 3. (Finishing the heap construction)
Predicate $\text{consistency}(i)$ is true when the heap values of the children of $i$ stored at $i$ are the same as the heap values stored at the corre-
spending children. When the root and its children are in state $M$ and consistency($r$) is true, the root changes its state to $P$ and executes macro $\text{init}.P(r)$ (Action $MP$). Eventually, every node changes its state from $M$ to $P$. This $P$ wave eventually reaches the leaves. The root initiates the BST construction when the root and its children are in $P$, i.e., the root can start the next phase even if not all nodes of the tree are in $P$ state.

Starting from the clean configuration presented in Figure 8.4(a), after executing the $\text{Heap}$ module when the root and its children are in state $M$, a possible configuration is given in Figure 8.9(a). The root, when surrounded by $M$ state children, changes its state to $P$ (Figure 8.9(b)). For each node we show the variable $s$, the state $S$, $lHV$, $HV$, $rHV$, and $down$. Symbol $b$ means $\bot$.

(a) Root and its children are done. (b) $P$ wave starts from the root.

Figure 8.9: The root and its children are done executing Module Heap.

From the root and down the tree, each node enters $P$ state and executes $\text{init}.P(i)$ (action $MP$). As an example, starting from the clean configuration presented in Figure 8.4(a), and after executing the $\text{Heap}$ module when $P$ wave had reached the leaf nodes, a possible configuration is given in Figure 8.10.

Figure 8.10: All the nodes are in $P$ state

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
We defined various abnormal predicates to characterize different types of local inconsistencies at a node during the heap construction. If any of these predicates is true at a node, then the only enabled action at that node will be $aCm$. This action when executed changes the state of the node to $C$.

8.5 Proof of Correctness of Algorithm Heap

We prove that starting from a normal starting configuration, the state of every node eventually becomes $B$ (for non-leaf nodes) or $M$ (for leaf nodes) (Property 8.5.1). Any internal node $i$ sets its heap value and changes its state from $B$ to $M/M^\text{left}/M^\text{right}$ (Property 8.5.2). When the root $r$ is in $M$ state, then $HV.r$, $lHV.r$, and $rHV.r$ hold the maximum value in the entire tree, in its left subtree, and in its right subtree, respectively (Property 8.5.3 for $i = r$). We conclude that, starting from a normal starting configuration, in at most $4h + 3$ rounds, the tree will satisfy the heap property and all nodes will be in state $P$ (Lemma 8.5).

**Property 8.5.1** Starting from a normal starting configuration, the following properties hold:

(i) If $i$ is a non-leaf node such that $S.i \neq C$, in at most $h - h_i + 2$ rounds, $i$ will change its state from $C$ to $B$.

(ii) If $i$ is a leaf node such that $S.i \neq C$, in at most $h + 2$ rounds, $i$ will change its state from $C$ to $M$.

**Proof.** Follows by induction on $m = h - h_i$, $0 \leq m \leq h$ and Actions $BC$ and $BM$. □

If $i$ is a non-leaf node in state $M^\text{left}/M^\text{right}$, then $i$ holds in $lHV.i$ ($rHV.i$) the maximum value in its left (right) subtree. However, if $i$ receives a value smaller than its heap value from its parent, it will need to push that value to one of its children.

**Property 8.5.2** Starting from a normal configuration where a non-leaf node $i$ is in state $B$, it takes $2h_i + 1$ rounds for $i$ to set $HV.i$, $lHV.i$, and $rHV.i$ as the maximum heap value in $T_i$, the maximum value in its left, and the maximum value in its right subtree, respectively, and
change its state from $B$ to $M / M^{left} / M^{right}$. If $i$ changes its state from $B$ to $M^{left} / M^{right}$, in at most $h_i + 1$ rounds, $i$ will change its state to $M$.

**Proof.** By induction on $m = h_i$, $0 \leq m \leq h$. When $i$ is in $B$ state, it takes $h_i$ rounds for the leaves of $T_i$ to enter $M$ state, and $h_i$ rounds for the children of $i$ to enter $M / M^{left} / M^{right}$ state, and one round for $i$ to change from $B$ to $M / M^{left} / M^{right}$ (Action $BM^*$). After changing its state from $B$ to $M / M^{left} / M^{right}$, node $i$ executes the macro set_HVs$(i)$ that collects in $lHV.i$ and $rHV.i$ the maximum value in its left and right subtree, respectively, and in $HV.r$ the maximum value in $T_i$. If node $i$'s old $HV$ value is smaller, then it is stored in variable $down$ and sent down the corresponding subtree.

We prove by induction on $m = h_i$, $1 \leq m \leq h$ that once node $i$ is in state $M^{left} / M^{right}$, it takes at most $h_i + 1$ rounds for $i$ to change to $M$. Assume that for some non-root node $j$ that is the $d \in \{left, right\}$ child of its parent: $S.j = M \land S.p.j = M^d$. (Such a node $j$ is guaranteed to exist since the leaf nodes can only be in $M$ state.) Then it takes two rounds for the parent of $j$ to enter $M$ state (Action $1rM$ is executed at node $j$. That makes Action $M^{lr}M$ enabled at the parent of $j$).

The worst case for $i$ is when there exists a path from $i$ to some of the leaves of $T_i$ such that any intermediate internal node has to push a value towards some descendant on that path and it has to wait for the descendant node to move before he moves. In at most $h_i + 1$ rounds, the value of $down$ reaches a node that accepts it without sending further down. □

If $i$ is in $M$ state and the action $MM^*$ is not enabled at $i$ (the parent has nothing to send), then $i$ holds in $lHV.i$ ($rHV.i$) the maximum value in its left (right) subtree, and in $HV.i$ the maximum value of $T_i$.

If $i$ is in $M$ state and Action $MM^*$ is enabled (some value has been sent by the parent and need to be properly placed in $T_i$), then in one round $i$ will push the value down and hold in $lHV.i$ ($rHV.i$) the maximum value in its left (right) subtree, and in $HV.i$ the maximum value of $T_i$.

**Property 8.5.3** For any node $i$, if $S.i = M$ and there are no values to be pushed down the tree $T_i$ (either $i$ is the root or none of its ancestors has a smaller value to push towards $i$) then $i$ remains in state $M$ until it is enabled to change to $P$, and it will take at most
$h - h_i + 1$ rounds for $i$ to change to $P$.

**Proof.** If a non-root node $i$ is in $M$ state, it is its left (right) child of its parent, and its parent is in $M/M^{\text{right}}$ ($M/M^{\text{left}}$) state, then $i$ is in HO and it will not change its state.

Since the root has no ancestors to push values towards it, once it enters $M$ state, it will not change to $M^{\text{left}}/M^{\text{right}}$. When the children of the root change to $P$, the root changes to $P$ in one round (Action $MP$). To show that it takes $h - h_i + 1$ rounds for any node $i$ to enter state $P$ (Action $MP$) we can use the induction on $m = h - h_i$, $0 \leq m \leq h$. □

**Lemma 8.5** Starting from a normal starting configuration, it takes at most $4h + 3$ rounds to heapify the tree.

**Proof.** Follows from Properties 8.5.1, 8.5.2, and 8.5.2. □
CHAPTER 9

BINARY-SEARCH-TREE CONSTRUCTION

In this chapter we present Algorithm BST, followed by its proof of correctness (Section 9.3).

9.1 BST Problem Specification

A snap-stabilizing algorithm guarantees that the system always behaves according to its specification provided some processor initiated the protocol.

Specification 9.1 (BST Problem) A protocol P is considered as a BST algorithm, if and only if the following conditions are true: (i) Any computation initiated by the root terminates in finite time. (ii) When the computation terminates, the values in the tree satisfy the BST property.

Remark 9.1 To prove that a BST algorithm is snap-stabilizing, we have to show that every execution of the protocol satisfies the following two properties: (i) starting from any configuration, the root eventually executes an initialization action. (ii) Any execution, starting from this action, satisfies Specification 9.1.

Being snap-stabilizing gives our algorithms a unique feature — they always behave as expected by their specifications. It should be noted that a self-stabilizing algorithm is guaranteed to satisfy the desired specification only in a finite time.

In the context of the BST problem, in a self-stabilizing BST solution, if the root initiates a BST computation, it is not guaranteed that the tree will become a BST when the computation terminates. If the computation is repeated (a bounded but unknown number of times), the self-stabilizing algorithm guarantees that eventually, the tree will become a BST. The proposed snap-stabilizing solution achieves a much better solution than the above. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ensures that when a BST computation initiated by the root terminates, the tree is a BST. Thus, we do not need to repeat the computation unless the application program demands repeated sorting of the values in the tree.

9.2 Binary-Search-Tree Algorithm

Given a binary tree where every node holds one key (value) drawn from an arbitrary set of real values, we design a snap-stabilizing distributed algorithm to arrange the values in the tree to obtain a binary search tree.

Under this space constraint, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor $i$ requires $O(\log s_i)$ bits where $s_i$ is the size of the subtree rooted at $i$. So, the root uses $O(\log n)$ bits.

The BST construction works as follows. First, the values in the tree are re-arranged as a heap (we implement a MaxHeap but a MinHeap is equally possible). For any node $i$, $i$ holds the maximum value among the values in $T_i$, the subtree rooted at $i$ (the maximum value is in the root node). For short, we write $\text{MHo}$ instead of MaxHeap order.

Based on the heap arrangement, the root collects values in decreasing order and delivers them to each node in the tree (a sequential, pipelined delivery of sorted values in decreasing order). The biggest value held by the root is sent to the leftmost leaf, followed by the second biggest, and so on, in a left-parent-right order. Given a 15-node tree and 15 values (Figure 9.1(a)), the unique BST built on those values is presented in Figure 9.1(b).

![Figure 9.1: Building a BST from a simple binary tree](image)

Our BST construction is transparent to the changes (addition or removal of notes) in the
tree structure. If such changes occur, then the algorithm will incorporate the changes "on
the fly" by nodes either entering an abnormal situation with respect to their new neighbors,
or by completing the current cycle and restarting a new cycle with added/deleted values.
We assume that after the add/remove operations/queries are executed, our algorithm will
be initiated by the root and a new BST tree will be constructed in $O(n)$ rounds. This makes
the lower bound of $\Omega(n)$ under the constraints considered in this work higher than that of
the usual functions (e.g., find, insert, and delete) for a BST.

9.2.1 Algorithm $BST$

In this section, we describe the data structures used, followed by a detailed explanation
of how the algorithm works when the initiator (the root process) starts the algorithm until
the values are arranged in the tree such that it becomes a BST.

The variables used by some node $i$ are described below. The sorted value $SV.i$ will
contain the final sorted value at the end of the algorithm. $tSV.i$ is used to store a temporary
sorted value. The heap value $HV.i$ is the result of the first phase (Algorithm $Heap$). The
new heap value $nHV$ is calculated and used in Algorithm $BST$. Variable $tmp$ is used for
storing the new internal value for an internal node that has a left subtree.

Algorithm $BST$ needs to maintain the size of the subtrees rooted at each node. This
size variable $s.i$ for node $i$ is computed in Algorithm $Heap$ and used in Algorithm $BST$. For
some node $i$, the variable $s.i$ requires $O(\log n_i)$ space where $n_i$ represents the total number
of descendants of $i$ in the subtree $T_i$ rooted at $i$. In the worst case (the root node) the space
complexity is $O(\log(n))$. Therefore the BST construction requires $O(\log n)$ space.

Algorithm $BST$ uses a total of seven states (see Figure 9.2). Besides the six states used
by Algorithm $Heap$ ($C$, $B$, $M$, $M^{left}$, $M^{right}$, and $P$), state $T$ (termination) is necessary.

Figure 9.2: The seven states used by the algorithm.
Some abnormal configurations can be locally detected by the processors. This local
detection is implemented using the abnormal predicates in Algorithm BST. Starting from
an abnormal configuration, an execution not necessarily will bring the system to a normal
starting configuration, but to a normal configuration. Starting from a normal configuration
where the root is able to execute the initialization action with no delay, the tree will become
a BST in $O(n)$ rounds. In general, the worst delay is $O(n)$ rounds because the worst initial
configuration is the one where no node has any of the abnormal predicates enabled, but
there is a node with an incorrect $tSV$ value (that does not match any $V$ values). Thus, the
abnormal configurations do not increase the asymptotic time bound. So, starting from any
configuration, the tree will become a BST in $O(n)$ rounds.

Algorithm 9.2.1 ModuleSort

Predicates

$\text{abnormal}_P(i, d) ::$ is true when the node, in state $P$, either has some variables with
abnormal values or is in abnormal situation with some neighbor (parent or child).

$\text{abnormal}_P(i, d) \equiv ab.P(i) \lor (p.i = \perp \land ab.P.rt(i)) \lor (p.i \neq \perp \land ab.P.nonrt(i)) \lor (D.i \neq \perp \land ab.P.nonlf(i)) \lor (D.i = \perp \land ab.P.If(i))$

$ab.P(i) \equiv s.i < 0 \lor \text{dir.i} \notin \{\perp, left, right\}$

$ab.P.rt() \equiv (s.r = 0 \land HV.r \neq \perp) \lor (HV.r = \perp \land s.r > 0)$

$ab.P.nornt(i) \equiv S.p.i \notin \{M, P, T\} \lor (d = left \land HV.p.i \notin \{HV.i, max(lHV.i, rHV.i)\}) \lor (d = right \land rHV.p.i \notin \{HV.i, max(lHV.i, rHV.i)\})$

$ab.P.nonlf(i) \equiv \exists j \in D.i(S.j \notin \{P, T\} \lor s.j + 1 \leq s.j) \lor (HV.i = \perp \land (lHV.i \neq \perp \land HV.left.i = \perp)) \lor (right.i \neq \perp \land rHV.i \neq \perp \land HV.right.i = \perp) \lor \neg s\_consistent(i)$

$ab.P.If(i) \equiv (lHV.i \neq \perp \lor rHV.i \neq \perp) \lor s.i > 1 \lor \text{down.i} \neq tSV.i \lor (s.i > 0 \land \text{down.i} \neq \perp)$

$s\_consistent(i) ::$ is true when $s.i$ is consistent with the variables $s$ of its children

The basic idea of the algorithm is as follows: The algorithm runs in two phases. The
root initiates the BST computation by starting a heapify process (shown as Algorithm Heap
\[ \text{s.con} \text{istent(i)} \equiv (s.i = 0 \land (r.\text{e}ft. \text{i} = \bot \lor s.\text{right} . \text{i} = 0)) \land (l.\text{e}ft. \text{i} = \bot \lor s.\text{left} . \text{i} = 0)) \lor (s.i > 0 \land((d.\text{own} . \text{i} = \bot \lor s.\text{ent}\_\text{sorted}(i)) \land s.i = s.\text{left} . \text{i} + s.\text{right} . \text{i} + 1) \lor (m.\text{y} . \text{t} \text{u}r \text{n} . \text{i}) \land s.i = s.\text{left} . \text{i} + s.\text{right} . \text{i} \lor (d.\text{own} . \text{i} = \bot \lor s.\text{ent}\_\text{sorted}(i)) \land \neg m.\text{y} \text{t} \text{u}r \text{n} . \text{i} \land s.i - 1 = s.\text{left} . \text{i} + s.\text{right} . \text{i}))) \]

\text{abnormal(T(i)) ::} \text{ when the node, in state } \text{T}, \text{ either has some variables with abnormal values or is in abnormal situation with some neighbor (parent or child).}

\text{abnormal(T(i)) } \equiv \neg \text{done(i)} \lor (D.i \neq \bot \land \forall j \in D.i \text{S}.j \neq T)

done(i) :: \text{ is true when the node is done executing the current BST cycle.}

done(i) \equiv s.i = 0 \land H.V.i = \bot \land t.S.V.i \neq \bot

\text{get.sorted(i) :: is true when the node is ready to receive a sorted value from its parent.}

\text{get.sorted(i)} \equiv s.i > 0 \land d.i . p. = d \land d.\text{own}. p.i \neq \bot \land d.\text{own}. i \neq d.\text{own}. p.i

\text{move.up.HV(i) :: is true when the node heap value was taken by its parent.}

\text{move.up.HV(i)} \equiv H.V.i \neq \bot \land ((d = \text{left} \land H.V.i \neq l.H.V.p.i) \lor (d = \text{right} \land H.V.i \neq r.H.V.p.i))

\text{new.sorted()} :: \text{ is true when the root has generated a new sorted value.}

\text{new.sorted()} \equiv H.V.r \neq \bot \land d.\text{own}. r = \bot \land s.r > 0 \land \text{consistency(r)}

\text{my.turn(i) :: is true when it is the node turn to store its sorted value (currently stored in down.i).}

\text{my.turn(i)} \equiv t.S.V.i = \bot \land (r.\text{e}ft. i = \bot \lor s.\text{right} . i = 0) \land \neg (d.\text{own} . i = \bot \lor s.\text{ent}\_\text{sorted})

\text{sent.sorted(i) :: is true when the node has sent the sorted value it held to one of its children (to its target node).}

\text{sent.sorted(i)} \equiv d.\text{own} . i \neq \bot \land d.i \neq \bot \land d.\text{own} . i = d.\text{own} . (d.i). i

\text{Macro move.HVs(i) :: selects the child } d.hv \in \{\text{left}, \text{right}\} \text{ with the maximum heap value.}

\text{dhv.i} = \text{MAX}(i)

\text{if (dhv.i = left} \land \text{left}.i \neq \bot \land l.H.V.i \neq \bot) \text{ then}

// the maximum value is in the left subtree so change H.V.i to that value and select

// in l.H.V.i the maximum value left in the left subtree

H.V.i = l.H.V.i

tmp.i = \text{MAX}(left.i)

\text{if tmp.i = left then}

\text{if down.i < l.H.V.left.i} \text{ then } l.H.V.i = l.H.V.left.i \text{ else } l.H.V.i = d.\text{own} . i

\text{if tmp.i = right then}

\text{if down.i < r.H.V.left.i} \text{ then } r.H.V.i = r.H.V.left.i \text{ else } l.H.V.i = d.\text{own} . i

\text{in Chapter 8) to create a maxheap of the tree. Then the root initiates the second phase (shown as Module } \text{Sort}. \text{ During this phase, the values are placed in the nodes in the BST}
if (dhv.i = right ∧ right.i ≠ ⊥ ∧ HV.i ≠ ⊥) then
  // the maximum value is in the right subtree so change HV.i to that value and select
  // in rHV.i the maximum value left in the right subtree
  HV.i = rHV.i
  tmp.i = MAX(right.i)
  if tmp = left then
    if down.i < lHV.right.i then rHV.i = lHV.right.i else rHV.i = down.i
  if tmp = right then
    if down.i < rHV.right.i then rHV.i = rHV.right.i else rHV.i = down.i

{Program for the root node r}

rP1 \neg abnormal.P(r, ⊥) ∧ S.r = P ∧ new_sorted() →
down.r = HV.r; s.r = s.r - 1; HV.r = ⊥
  if (lHV.i ≠ ⊥ ∨ rHV.i ≠ ⊥) then move.HVs(r)

rP2 \neg abnormal.P(r, ⊥) ∧ S.r = P ∧ sent.sorted(r) ∧ ¬my.turn(r) → down.r = ⊥

rP3 \neg abnormal.P(r, ⊥) ∧ S.r = P ∧ my.turn(r) →
tSV.r = down.r
  if SV.r ≠ tSV.r then SV.r = tSV.r
  if left.r ≠ ⊥ then dir.r = left else dir.r = ⊥
  down.r = ⊥

PT \neg abnormal.P(r, ⊥) ∧ S.r = P ∧ done(r) ∧ ∀j∈D, rS.j = T → S.r = T

aCb (S.r = P ∧ abnormal.P(r, ⊥)) ∨ (S.r = T ∧ abnormal.T(r)) → S.r = C

{Program for an internal node i that is the d child of its parent, d ∈ {left, right}}

iP1 \neg abnormal.P(i) ∧ S.i = P ∧ get_sorted(i) ∧ (down.i = ⊥ ∨ sent.sorted(i)) →
down.i = down.p.i; s.i = s.i - 1
  if moveup.Hv(i) then HV.i = ⊥
    if (lHV.i ≠ ⊥ ∨ rHV.i ≠ ⊥) then move.HVs(i)

iP3 \neg abnormal.P(i) ∧ S.i = P ∧ my.turn(i) →
tSV.i = down.i
  if SV.i ≠ tSV.i then SV.i = tSV.i
  if left.i ≠ ⊥ then dir.i = left else dir.i = ⊥
order, placing the highest value first, the second highest value next, and so on. As the maxheap has been created in the previous phase, the root holds the maximum value of the tree. This highest value is sent to the rightmost node (say, $i$) of the tree. The destination of the second highest value (say, $second$) is dependent on if $i$ is a leaf or an internal node. If $i$ is a leaf node, then $second$ is sent to the parent of $i$ (say, $j$). Then the third highest value (say, $third$) will be sent to the left child of $j$ (if present) or to the parent of $j$.

If $i$ is an internal node, then $second$ goes to the left child of $i$. Thus, values are placed in the tree following a right-parent-left order.

The algorithm will be similar if we have constructed a minheap instead of the maxheap. In that case, in the second phase, the values will be placed following a left-parent-right order. From now on, heap will imply maxheap. If a node $i$ satisfies the maxheap property with respect to its parent and children, we say $i$ is in heap order or in HO in short.
The interface between the two layers (application and BST) at a node $i$ is implemented by two variables: input value to the sorting protocol $V_i$ and the final or output sorted value $SV_i$. However, every time the BST protocol runs, we do not want to disturb the application layer by writing (or overwriting) the value of $SV_i$ unless the value has changed. So, when the BST protocol terminates, $i$'s sorted value is first placed in $tSV_i$. Then $tSV_i$ and $SV_i$ are compared. The value of $tSV_i$ is copied into $SV_i$ only if the values are different (see Actions $rP3$, $iP3$, and $lP1&3$ of Module $Sort$).

At the end of the heap construction, every node changes its state from $M$ to $P$ and executes the macro $init.P(i)$. In this macro, every non-leaf node $i$ sets the variable $dir.i$ to point to the child that will receive the sorted value from the root. Recall that the sorted values are placed in right-parent-left order.

Every node (including the root) will receive a sorted value from the root and send its heap value to the root. These two actions are executed concurrently. Upon completion of the heap, the root holds the maximum (heap) value of the entire tree, its children hold the maximum (heap) values of their subtrees, and so on. The above heap property is exploited in the BST construction. The root first sends out its own heap value to the rightmost place in the tree. The root then gets the second highest value of the tree easily (in constant steps) from one of its children. So, the concurrency of the two main tasks — sending the sorted value to the proper place and moving the heap values upward toward the root — are achieved by using the heap property. That is the reason of using the heap phase as a pre-processing phase of the BST construction.

When a sorted value sent to a node belongs to that node (i.e., it is the node's sorted value), it is stored in $tSV$. A node is done sorting if all nodes in its subtree (including itself) received their final sorted values. This is checked in the predicate $done$. When a node is done, it changes its state to $T$. Obviously, this wave of state change from $P$ to $T$ starts from the leaves and ends at the root. When the root changes its state to $T$, the algorithm terminates. In the following, we describe a normal execution of module $Sort$:

4. (Select sorted values for all nodes) Predicate $new.sorted()$ is true if the root still has values to sort: $HV.r \neq \perp$, either it just started or the previous sorted value has been delivered ($down.r = \perp$), there are nodes that need more sorted values ($s.r > 0$), and it has
consistency with its children \((\text{consistency}(r) = \text{true}).\)

If the root is in \(P\) and Predicate \(\text{new\_sorted()}\) is \(\text{true},\) the only enabled action is Action \(rP1.\) So, it will eventually be executed. The current \(HV.r\) value is moved into \(\text{down}.r,\) \(s.r\) is decremented, and \(HV.r\) becomes \(\bot.\) Then the larger of the heap values of one of its children is moved in \(HV.r\) by executing the macro \(\text{move\_HVs}(r).\) That will enable Action \(rP1\) again.

5. \((\text{Receive sorted value and/or collect heap value})\) Although these two actions are executed concurrently, we present them separately below:

5.1 \((\text{Receive sorted value})\) We first define a \(\text{target node}\) for some node. For some node \(i,\) if the condition \(s.i > 0 \lor (s.i = 0 \land (\text{left}.i \neq \bot \land s.\text{left}.i = 1)\) is \(\text{true},\) then there exists a unique node \(j\) to which \(\text{down}.i \neq \bot\) will be delivered (either \(j = i\) or \(j\) is one of the children of \(i\)). We call node \(j\) the \(\text{current target}\) of node \(i.\) \(\text{dir}.i\) holds the value \(j.\)

We use the following predicates in this part of the algorithm:

Predicate \(\text{sent\_sorted}(i)\) is \(\text{true}\) if the non-root node \(i\) has previously received another value from its parent \(p.i\) and it has already delivered it.

Predicate \(\text{my\_turn}(i)\) is \(\text{true}\) if it is the turn of node \(i\) to collect its sorted value. Node \(i\) has no current sorted value \((tSV.i \neq \bot),\) either it has no right subtree \((\text{right}.i = \bot)\) or is full \((s.\text{right}.i = 0),\) and has a value \((\text{down}.i \neq \bot)\) that was not taken by any of the children of node \(i\) (Predicate \(\text{sent\_sorted}(i)\) is \(\text{false}).\)

Predicate \(\text{get\_sorted}(j)\) is \(\text{true}\) if \(j,\) the \(d \in \{\text{left, right}\}\) child of its parent \(i,\) is allowed to copy in \(\text{down}.i\) the value stored at its parent, \(j\) still needs sorted values \((s.j > 0),\) it is the current target of node \(i\) \((\text{dir}.i = d),\) and the sorted value held by \(i\) is a new one \((\text{down}.i \neq \text{down}.j).\)

During the BST construction, if Predicate \(\text{my\_turn}(i)\) is \(\text{true},\) the target of node \(i\) is \(i\) itself. Otherwise, for a non-leaf node \(i,\) the target \(j\) of node \(i\) is one of the children of \(i\) that is allowed to copy into \(\text{down}.j\) the value stored at \(\text{down}.i\) if either Action \(iP1\) or \(iP1\&3\) is enabled and executed. We now consider the three types of target node \(j\) of node \(i\) (root, internal, and leaf) below:

[Root]) The target node is the root itself \((i = j = r).\) Then \(\text{my\_turn}(r)\) is \(\text{true}\) and the only enabled action is Action \(rP3.\) The root moves \(\text{down}.r\) into \(tSV.r,\) updates \(SV.r\)
if necessary, and selects its left child (if exists) as its current target (by changing \texttt{dir.r} to \texttt{left}), and sets \texttt{down.r} to \texttt{⊥}.

[\textbf{Internal}] The target node \texttt{j} is an internal node. We have two cases for \texttt{j}:

\textbf{1)} If \texttt{my.turn(j)} is \texttt{true}, then the only enabled action for \texttt{j} is Action \texttt{iP3} which is similar to Action \texttt{rP3}.

\textbf{2)} If \texttt{my.turn(j)} is \texttt{false}, then the only possible enabled action for \texttt{j} is Action \texttt{iP1}. \texttt{iP1} is enabled if \texttt{get.sorted(j)} is \texttt{true} and the condition \texttt{down.j} = \texttt{⊥} \lor \texttt{sent.sorted(j)} are \texttt{true}.

Condition \texttt{down.j} = \texttt{⊥} \lor \texttt{sent.sorted(j)} is \texttt{true} if either \texttt{j} has never received a value from \texttt{i} (\texttt{down.j} = \texttt{⊥}), or has previously received another value from \texttt{i} and has already delivered it (\texttt{sent.sorted(j)} is \texttt{true}).

When Action \texttt{iP1} is performed, \texttt{down.i} is copied into \texttt{down.j}, \texttt{s.j} is decremented, and \texttt{j} checks if it has to give up its heap value to its parent (Predicate \texttt{moveup.hv(j)} is explained below).

[\textbf{Leaf}] The target node \texttt{j} is a leaf node. Since the only value the leaf is allowed to receive is its own sorted value, the target of \texttt{j} is \texttt{j} itself. If \texttt{get.sorted(i)} is \texttt{true}, the only action enabled at \texttt{j} is \texttt{iP1&3}, so it eventually gets executed.

\textbf{5.2 (Collect heap value)} Predicate \texttt{moveup.hv(i)} is \texttt{true} for some node \texttt{i} if \texttt{i}'s heap value (\texttt{HV.i} \neq \texttt{⊥}) was taken by \texttt{p.i} as its heap value. In that case, \texttt{i} selects the larger of the heap values of its children as its next heap value. Variable \texttt{dhv.i} indicates which child (heap value) will be selected. We now need to distinguish three cases.

\textit{Case 1} [\texttt{HV.i} = \texttt{⊥}] \texttt{i} waits until \texttt{done(i)} becomes \texttt{true} so that it can change to state \texttt{T}.

\textit{Case 2} [\texttt{HV.i} \neq \texttt{⊥} \land \texttt{IHV.i} = \texttt{⊥} \land \texttt{rHV.i} = \texttt{⊥}] If node \texttt{i} is the root node \texttt{r}, then it has to wait until Action \texttt{rP1} becomes enabled and gets executed. Then \texttt{HV.i} becomes \texttt{⊥} and \textit{Case 1} becomes applicable.

If \texttt{i} is a non-root node, then when \texttt{moveup.hv(i)} is \texttt{true}, the heap value \texttt{HV.i} \neq \texttt{⊥} is moved up the tree from node \texttt{i} to its parent \texttt{p.i} (action \texttt{iP1}, \texttt{iP4}, \texttt{iP1&3}, or \texttt{iP4} is enabled and executed). \texttt{HV.i} becomes \texttt{⊥} and \textit{Case 1} becomes applicable.

\textit{Case 3} [\texttt{HV.i} \neq \texttt{⊥} \land (\texttt{IHV.i} \neq \texttt{⊥} \lor \texttt{rHV.i} \neq \texttt{⊥})] Node \texttt{i} is a non-leaf node, and there exists a unique node \texttt{j} (decided in macro \texttt{move.HV.s(i)}) that will move its heap value to \texttt{i}'s heap value when one of Actions \texttt{rP1}, \texttt{iP1}, \texttt{iP4}, \texttt{iP1&3}, and \texttt{iP4} is executed. Node \texttt{j} is one
of the children of \( i \) and is called the current sink of node \( i \).

If node \( i \) is the root node \( r \), then it has to wait until Action \( rP1 \) becomes enabled and gets executed. Then macro \( \text{move.HVs}(r) \) is executed and \( HV.i \) receives the larger of the heap value of its children. Either Case 2 or Case 3 becomes applicable then.

If \( i \) is a non-root node, then if \( \text{moveup.hv}(i) \) is true, the heap value \( HV.i \neq \perp \) is moved up the tree from node \( i \) to its parent \( p.i \) and \( i \) executes macro \( \text{move.HVs}(i) \) (Action \( iP1 \) or \( iP4 \) is enabled and executed). Either Case 2 or Case 3 becomes enabled.

For example, starting from a configuration where the root and its children are in state \( P \), after executing the action \( rP1 \), we obtain the configuration shown in Figure 9.3(a). Now its right child has to execute \( iP1 \) before the root is able to move again and execute \( rP2 \). Also its left child has to execute \( iP4 \) before the root can execute \( rP1 \). Once both children execute, we obtain the configuration as shown in Figure 9.3(b). For each node, we show the variable \( s \), the state \( S \), \( tHV \), \( HV \), \( rHV \), \( down \), and \( dir \). Symbol \( b \) means \( \perp \).

![Diagram](image)

(a) Root generates a sorted value  
(Artion \( rP1 \))  
(b) Right node gets the value,  
left node changes its \( HV \)

Figure 9.3: Some state ending the execution of Algorithm Heap by the root and its children

6. (Sets its own sorted value and adjusts the direction for the future sorted values) If \( \text{my.turn}(i) \) is true, \( i \) collects its sorted value and adjusts the direction of sorted values toward its left subtree, if it exists. Otherwise, \( \text{dir.i} \) is set to \( \perp \).

7. (Terminating the BST) Predicate \( \text{done}(i) \) is true when node \( i \) has \( HV.i = \perp \), does not need more sorted values from the root, and has its currently sorted value \( tSV.i \neq \perp \).

When a leaf node \( i \) is done receiving the sorted values (\( \text{done}(i) \) is true), it changes its state from \( P \) to \( T \). When a non-leaf node \( i \) is done receiving its sorted values (predicate
**done(i) is true** and all its children are in state T, i changes its state from P to T (Action $PT$ is enabled).

Starting from the configuration presented in Figure 8.10, after executing Module Sort when $T$ wave had reached the root node, the final configuration is given in Figure 9.4.

![Figure 9.4: The tree is a BST](image)

We defined various abnormal predicates to characterize different types of local inconsistencies at a node during the BST construction. If any of these predicates is true at a node, then the only enabled action at that node will be $aCb$. This action when executed changes the state of the node to $C$.

### 9.3 Proof of Correctness for Algorithm BST

In this section, we first present the proof of correctness assuming the weakly fair daemon. (A daemon is weakly fair if a continuously enabled process will be eventually chosen by the daemon.) Later in Section 9.3.3, we show that the algorithm works under the unfair daemon as well.

We first show a lower bound of $\Omega(n)$ on the time complexity for the BST problem under the constraint as discussed earlier (Lemma 9.1).

Next, we show how the algorithm corrects any abnormal configuration into a normal configuration in finite number of rounds. Considering faulty networks, the system may start in an abnormal configuration where there exists at least one abnormal processor. We prove that if some node $i$ is abnormal, then $S.i$ becomes $C$ in at most one round (Property 9.3.1). Using this result, we show that if $S.i = C$, all the nodes in the subtree rooted at $i$, $T_i$
change to $C$ in $O(h_i)$ rounds (Property 9.3.2). Then in $O(h)$ rounds the system reaches a configuration which does not contain any local problem and the behavior of the protocol is now almost as the normal behavior (the result is in Lemma 9.3. We conclude that the delay (the time needed for the root to execute the initialization action) of our algorithm is $O(n)$ rounds (Lemma 9.2).

Once we establish the finite round delay (as above), our remaining obligation is to show that starting from a normal starting configuration, the tree will satisfy the BST property in finite rounds.

Note that the guarded actions of Module Sort for each process (root, internal, and leaf node) are mutually exclusive. So, at any time during the BST construction, at most one of the root actions is enabled, at most one of any internal node actions is enabled, and at most one of any leaf node actions is enabled. This was done to implement a sequential, pipelined delivery of sorted values in decreasing order.

The root will continue sending the sorted values (via $d_{own.r}$) in descending order as long as there exists a target node for a value ($new_{-sorted}()$ will be true as long as $s.r > 0$) (Property 9.3.6). The value of $d_{own.r}$ follows a path of current target nodes, starting from the root and ending at some node in the tree. When the root takes the heap value of its sink node to make it a sorted value, in at most one round, the child adjust its heap value to one of its children (Property 9.3.11). In at most $n$ rounds, the root is done generating sorted values (Property 9.3.13), and in at most additional $h$ rounds every node in the system receives its sorted value (Lemma 9.3), and enters state $T$. Once the root enters $T$ state, the BST construction is done.

Finally, we prove that starting from an arbitrary configuration where the nodes have distinct internal values drawn from an arbitrary set, our algorithm arranges them in a BST order in $O(n)$ rounds (Theorem 9.3.14).

Theorem 9.3.14 and Lemma 9.1 imply that the proposed BST algorithm is time optimal.

**Lemma 9.1** Under the space constraint that the maximum number of items that can be stored at any time at any processor is constant (i.e., independent of $n$), the lower bound on the time complexity for arranging $n$ values in a given tree in a distributed manner such that
the tree becomes a binary search tree (BST) is $\Omega(n)$.

**Proof.** Assume that all the values larger (respectively, smaller) than the root's value are currently in the left (respectively, right) subtree of the root. Then $n - 1$ values have to pass by the root to move to their right place in the BST. As the root has a constant memory, it will require the root to execute at least $n$ actions to move those values. □

9.3.1 Abnormal Configurations and Delay

In this subsection, we prove that starting from any configuration, the root executes the initialization action in $O(n)$ rounds (Lemma 9.2).

Considering faulty networks, the system may start in an abnormal configuration where there exists at least one abnormal processor. We prove that if some node $i$ is abnormal, then $S.i$ becomes $C$ in at most one round (Property 9.3.1). Using this result, we show that if $S.i = C$, all the nodes in the subtree rooted at $i$, $T_i$ change to $C$ in $O(h_i)$ rounds (Property 9.3.2). Thus, if $i = r$, then in $O(h)$ rounds the tree reaches a normal starting configuration. This leads to the next result that if $i \neq r$, in at most $O(h)$ rounds, the system reaches a normal configuration (Property 9.3.3). We conclude that the delay (the time needed to reach a normal configuration) of our algorithm is $O(h)$ rounds (Lemma 9.2).

**Property 9.3.1** If $i$ is in an abnormal state (i.e., some predicate abnormal is true at $i$), $S.i$ will become $C$ in at most one round.

**Proof.** The guards of all actions except $aCm$ and $aCb$ will be disabled at $i$ when $i$ is in abnormal state. So, either $aCm$ or $aCb$ will be executed in the current round and $S.i$ will change to $C$. □

**Property 9.3.2** If $i$ is in $C$, then all other nodes in the subtree $T_i$ will also be in state $C$ in $O(h_i)$ rounds where $h_i$ is the height of $T_i$.

**Proof.** Consider any child $j$ of $i$. Since $S.i = C$, if $S.j$ is not already in $C$, one of the abnormal predicates is true at $j$ and either $aCm$ or $aCb$ is enabled. As explained in the proof of Property 9.3.1, $S.j$ becomes $C$ in at most one round. The property follows by induction on the height $h_i$ of the subtree $T_i$ and the fact that the left and right branches of the subtree can work in parallel. □
Property 9.3.3 If some node \( i \neq r \) has \( S.i = C \), then it may exist an ancestor \( k \) of \( i \) whose state \( S.k \) becomes \( C \) because of \( i \)'s state. If no ancestor of \( i \) changes its state to \( C \), then we consider \( k \) to be \( i \) itself. Then all nodes in \( T_k \) will be in state \( C \) in \( O(h_k) \) rounds.

Proof. Consider the path from \( k \) to \( i \) down the tree. By the property hypothesis, there must exist a pair of nodes \( p \) and \( q \) such that \( x = p.q \wedge S.x \neq C \wedge S.y = C \), and \( x \) is in an abnormal state w.r.t \( y \). So, by Property 9.3.1, \( S.x \) will become \( C \) and by Property 9.3.2, all nodes in \( T_x \) will be in \( C \). The property follows by repeated application of the above reasoning on the nodes on the path from \( i \) to \( k \) up the tree. \( \square \)

Lemma 9.2 The delay is in \( O(n) \) rounds.

Proof. If \( i = r \), by Properties 9.3.1 and 9.3.2, the system does not contain any locally detectable problem in \( O(h) \) rounds. Now the system behaves as if it was correctly initialized. From Lemma 9.3, the time complexity in this case is in \( O(n) \) rounds.

In the worst case, the root can execute the initialization action in \( O(h) + O(n) = O(n) \) rounds. \( \square \)

9.3.2 Proof of correctness for Algorithm BST

In this subsection, we prove that starting from a normal configuration where the root and its children are in state \( P \), the algorithm builds a BST in \( O(n) \) rounds (Lemma 9.3).

If a node \( i \) is in \( P \), then the variable \( s.i \) indicates the number of sorted values \( i \) will eventually receive for its subtree \( T_i \) (Property 9.3.4).

We also show that \( i \) sets its current sorted value (\( tSV.i \)) at most once (Property 9.3.5).

The root will continue sending the sorted values (via \( down.r \)) in descending order as long as there exists a target node for a value (\( new_sorted() \) will be true as long as \( s.r > 0 \)) (Property 9.3.6). When \( down.r \neq \perp \) is delivered to the current target node of the root, \( down.r \) becomes \( \perp \) again (Property 9.3.7).

The value of \( down.r \) follows a path of current target nodes, starting from the root and ending at some node in the tree. It takes at most one round for some node to take that value from its parent (Property 9.3.8). If the target of node \( i \) is \( i \) itself, it takes at most one round to collect the sorted value (Property 9.3.9). Otherwise, the target node is one of
i’s children, so it takes more rounds for that value and other values in the subtree \(T_i\) to be delivered and collected by nodes in \(T_i\) (Property 9.3.10).

When the root takes the heap value of its sink node to make it a sorted value, in at most one round, the child adjust its heap value to one of its children (Property 9.3.11). In at most \(n\) rounds, the root is done generating sorted values (Property 9.3.13), and in at most additional \(h\) rounds every node in the system receives its sorted value (Lemma 9.3), and enters state \(T\). Once the root enters \(T\) state, the BST construction is done.

**Property 9.3.4** In any normal configuration where \(S.i = P\), \(s.i\) represents the number of sorted values to arrange in \(T_i\).

**Proof.** Follows by induction on \(m = h_i\). For \(m = 0\), if \(s.i \in \{0, 1\}\) depending whether the node has collected its sorted value from its parent or not. For \(m > 0\), as long as Predicate \(s\text{-consistent}(i)\) is true, \(s.i\) represents how many nodes in \(T_i\) had not received a sorted value. These nodes can be from the left subtree, right subtree, or node \(i\) itself. □

**Property 9.3.5** In a normal execution, each node \(i\) sets its current sorted value \((tSV.i)\) at most once (Action \(rP3, iP3,\) or \(IP1\&3\) is executed at most once).

**Proof.** If \(tSV.i \neq \perp\), then Predicate \(my\_turn(i)\) is false. So, depending on the type of the node (root, internal, or leaf node), Action \(rP3, iP3,\) or \(IP1\&3\) is enabled. □

**Property 9.3.6** Starting from a normal configuration where the root and its children are in state \(P\) and \(s.r > 0\), in at most \(m\) rounds \((0 \leq m \leq h)\), Predicate \(new\_sorted()\) will become true, and in another round, at least \(m\) nodes will receive their sorted values.

**Proof.** If Predicate \(new\_sorted()\) is false, the system is in a normal configuration and \(s.r > 0\). This implies that the current value of \(down.r\) has not been delivered to the target node of \(r\). If the target node of \(r\) is \(r\) itself, in at most one round, \(down.r\) becomes the root’s sorted value. If the target node is not \(r\), but one of its children, say \(j\), it may take up to \(m\) rounds \((0 < m \leq h)\) for \(j\) to receive the value \(down.r\), i.e., for Predicate \(sent\_sorted(j)\) to become \(true\) (Property 9.3.10). So, in at most \(m + 1\) rounds, Predicate \(new\_sorted()\) is re-evaluated to \(true\), and in the meantime, \(m\) nodes have received their sorted values.
If Predicate `new_sorted()` is true, in at most one round the root generates in `down.r` a sorted value for some node in the tree (Action `rP1` is enabled).

**Property 9.3.7** If `down.r ≠ ⊥` and `s.r > 0`, then one of the following is true:
- if Predicate `my_turn(r)` is true, then in at most one round, `down` becomes the root's sorted value, then the root directs the flow to either its left child or none, and `down` is set to `⊥` (Action `rP3` is executed).
- if Predicate `my_turn(r)` is false and Predicate `sent_sorted(r)` is true, then in at most one round, `down.r` is pushed to one of the children (indicated by `dir.r`) and set to `⊥` (Action `rP2` is executed).

In both cases, `down.r` becomes `⊥`.

**Property 9.3.8** For any non-root node `i`, if node `i` expects a sorted value from its parent (Predicate `get_sorted(i)` is true), and either `i` has never received a sorted value from its parent (`down.i = ⊥`) or it has received a previous value that has already been delivered to the appropriate node (Predicate `sent_sorted(i)` is true), then in at most one round, the value sent by the root is received by node `i` (either Action `iP1` or `iP1&3` is executed).

**Property 9.3.9** For any non-root node `i`, if it is the turn of `i` to get its sorted value (Predicate `my_turn(i)` is true), then in at most one round, `down` becomes the node's sorted value (Action `iP3` or `iP1&3` is executed). If `i` is a non-leaf node, `i` directs the flow of `down` to either its left child or none (Action `iP3` is executed).

**Property 9.3.10** For any non-root node `i`, if node `i` has to deliver value `i` to its target node that is one of its children and has not done so (Predicate `sent_sorted(i)` is false), then `i` will deliver value `i` value in some number `m` of rounds where `1 ≤ m ≤ h_i`. In the meantime, `m` nodes in `T_i` will receive their sorted values.

**Proof.** By induction on `m =` the number of target nodes on the current flow path, starting from the target node of the root, that have Predicate `sent_sorted()` evaluated to `false`. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Property 9.3.11 For any non-root node $i$, if node $i$ is the sink node for its parent (Predicate $\text{moveup}_{hv}(i)$ is true), then in at most one round, $i$ takes the maximum heap value among its children (Macro $\text{moveHVs}(i)$ is executed) and some node in the subtree rooted at $i$ loses its heap value (the heap value of that node becomes $\bot$).

Proof. Either action $iP1$ or $iP4$ is enabled at some internal node $i$ and executed. Either action $iP1&3$ or $iP4$ is enabled at some leaf node $i$ and executed. □

Property 9.3.12 For any node $i$ in the tree, if $HV.i = \bot$, then every node $j$ in $T_i$ has $HV.j = \bot$.

Proof. By induction on $m = h_i$. □

Property 9.3.13 Starting from a normal configuration where the root and its children are in state $P$, in at most $n$ rounds, the root generates $n$ sorted values and $s.r$ becomes 0.

Proof. By Property 9.3.4, we know that the root has to generate $s.r = n$ sorted values. Property 9.3.6 shows the gap between two consecutive creations of sorted values. Since there are no more than $n$ sorted values, in no more than $n$ rounds, Predicate $\text{new\_sorted}()$ will be enabled $n$ times. □

Lemma 9.3 Starting from a normal configuration, where the root and its children are in state $P$, in at most $n + h$ rounds, every node in the system receives its sorted value and changes its state to $T$.

Proof. From the definition of a normal configuration and Property 9.3.13, in at most $n$ rounds, $s.r$ becomes 0. When $s.r = 0$, every node in the right subtree has $s.i = 0$. Then in at most $h$ rounds (Property 9.3.10), every node in the left subtree has $s.i = 0$ once the last value $\text{down.r}$ is received by the leftmost node in the tree.

When $s.r = 0$, $HV.r = \bot$ and every node $i$ in the tree has $HV.i = \bot$ (Property 9.3.12). So, every node, including the root, will change its state from $P$ to $T$ (Action $PT$ is executed). □
Theorem 9.3.14 Starting from an arbitrary configuration where \( n \) values are arranged in a binary tree, each node holding a single key value, Algorithm 9.2.1 arranges those \( n \) values such that the tree becomes a BST in \( O(n) \) rounds and requires \( O(\log n) \) space.

9.3.3 Unfair Daemon

In any round, the total number of actions executed by all processes is bounded. Since any execution of our algorithm has a bounded complexity in terms of steps (or actions), the total number of actions executed in a normal execution is bounded. Thus, the duration of a round cannot be extended forever by ignoring some enabled processes for an indefinite period of time.
CHAPTER 10

CONCLUSIONS

In this dissertation, we presented several self or snap-stabilizing algorithms for particular data structures.

We proposed an optimal self-stabilizing distributed algorithm for simultaneously activating non-adjacent processes on an oriented chain (Algorithm $SSDS$). We used Algorithm $SSDS$ to accomplish two tasks: local mutual exclusion and line sorting. We proposed two uniform, self-stabilizing, deterministic protocols on oriented chains: a time and space optimal solution to the local mutual exclusion problem (Algorithm $LMEC$), and a space and (asymptotic) time optimal solution to the distributed sorting problem (Algorithm $SORT_e$).

We extended Algorithm $SSDS$ to an asynchronous oriented ring with a distinguished node with some minor modifications, and we obtain general self-stabilization for simultaneously activated non-adjacent processes in an oriented ring with a distinguished process (Algorithm $SSDSR$). We used Algorithm $SSDSR$ to accomplish two tasks: local resource allocation and ring sorting. We proposed two uniform, self-stabilizing, deterministic protocols on oriented rings: a time and space optimal solution to the local resource allocation problem (Algorithm $LRAR$), and a space and (asymptotic) time optimal solution to the distributed sorting problem (Algorithm $SORT_r$).

We extended Algorithm $SSDS$ to an asynchronous rooted tree, and we obtain general self-stabilization for simultaneously activated non-adjacent processes in a rooted tree (Algorithm $SSDST$). We then gave two applications of Algorithm $SSDST$: a time and space optimal solution to the local mutual exclusion problem (Algorithm $LMET$) and a space and (asymptotically) time optimal solution to the min heap problem (Algorithm $HEAP$).

In proving the time complexity of sorting, we introduced the notion of pseudo-time,
similar to *logical time* introduced by Lamport [Lam78].

We presented the first snap-stabilizing distributed binary search tree (BST) algorithm. The proposed algorithm uses a heap algorithm (Algorithm *Heap*) as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem.

We expect that Algorithms *SSDS*, *SSDSR*, and *SSDST* can be used to obtain optimal space solutions for other problems in a rooted tree. For example, for broadcasting *m* messages, a solution based on Algorithm *SSDST* stabilizes in at most $4h + 2m - 5$ rounds (the root node executes *m* times). An interesting topic of future research is to find efficient self-stabilizing solutions (more efficient than the existing ones) to the existent problems in distributed computing as broadcasting, leader election, mutual exclusion, on particular topologies for which an optimal space algorithm for simultaneously activated non-adjacent processes we have presented. Another topic is to find more particular topologies of general use (especially interconnection networks) for which such an algorithm can be designed for.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Doina Bein

Home Address:
2355 Brockton Way
Henderson, NV 89074

Degrees:
Bachelor of Science, Computer Science, 1996
Al. I. Cuza University of Iasi, Romania

Master of Science, Computer Science, 1997
Al. I. Cuza University of Iasi, Romania

Master of Science, Computer Science, 2001
University of Nevada, Las Vegas

Dissertation Title: Distributed Stabilizing Data Structures

Dissertation Examination Committee:
Chairperson, Dr. Ajoy Kumar Datta, Ph.D.
Co-chairperson, Dr. Lawrence L. Larmore, Ph.D.
Committee Member, Dr. Vincent Villain, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Graduate Faculty Representative, Dr. Shahram Latifi, Ph.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.