Self-stabilizing sorting on linear networks

Lakshmi Visvanathan

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1996
SELF-STABILIZING SORTING ON LINEAR NETWORKS

by

Lakshmi Visvanathan
Bachelor of Science
Bharathidasan University
2001

A thesis submitted in partial fulfillment
of the requirements for the

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

Graduate College
University of Nevada, Las Vegas
May 2006

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

LAKSHMI VISVANATHAN

Entitled

SELF-STABILIZING SORTING ON LINEAR NETWORKS

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

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Self-stabilizing Sorting on Linear Networks

by

Lakshmi Visvanathan

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

A self-stabilizing system has the ability to recover from an arbitrary (possibly faulty) state to a normal state without any manual intervention. A self-stabilizing algorithm does not require any initialization. Starting from an arbitrary state, it is guaranteed to satisfy its specification in finite number of steps.

We propose a self-stabilizing distributed sorting algorithm on an oriented linear network with \( n \) nodes. Each node holds some initial value(s) drawn from an arbitrary set. We assume that we start with at most \( k \) items in the network. Each node has a local memory whose space is restricted to \( O(k \times L) \) where \( L \) is the maximum number of bits to store one item. A node may collect more than one value during the process of sorting. The stabilizing time for sorting is \( O(n) \) rounds where a round is the duration for all the enabled processes to execute at least one enabled step. We claim that our algorithm is self-stabilizing for the following reasons:

If any node starts in a faulty state, (meaning its value is not sorted with respect to its neighbors), the algorithm guarantees that the node will reach the legitimate state (where a legitimate state is a state in which the values are in sorted order) in a finite amount of time, and will remain in the legitimate state until another fault occurs. Each node repeatedly communicates with its neighbors to check if the values of its neighbors are sorted with respect to its own value. If the values are not in order, either the node or one of its neighbors will eventually be enabled to execute
so that in finite amount of time the values will be sorted.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td><strong>CHAPTER 1  INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>Related Work</td>
<td>2</td>
</tr>
<tr>
<td>Contributions</td>
<td>4</td>
</tr>
<tr>
<td>Outline of the Thesis</td>
<td>5</td>
</tr>
<tr>
<td><strong>CHAPTER 2  PRELIMINARIES</strong></td>
<td>6</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>6</td>
</tr>
<tr>
<td>Synchronous and Asynchronous Systems</td>
<td>7</td>
</tr>
<tr>
<td>Self-stabilization</td>
<td>8</td>
</tr>
<tr>
<td>Oriented Chains</td>
<td>10</td>
</tr>
<tr>
<td>Lower and Upper Bounds for Sorting on Non-Stabilizing Systems</td>
<td>10</td>
</tr>
<tr>
<td>Applications of Sorting</td>
<td>11</td>
</tr>
<tr>
<td><strong>CHAPTER 3  OUR ALGORITHM AND DATA STRUCTURES USED</strong></td>
<td>12</td>
</tr>
<tr>
<td>Variables</td>
<td>12</td>
</tr>
<tr>
<td>Predicates</td>
<td>13</td>
</tr>
<tr>
<td>Macros</td>
<td>15</td>
</tr>
<tr>
<td>The Code of our Algorithm</td>
<td>16</td>
</tr>
<tr>
<td>Explanation of our Algorithm</td>
<td>21</td>
</tr>
<tr>
<td><strong>CHAPTER 4  PROOF OF CORRECTNESS</strong></td>
<td>26</td>
</tr>
<tr>
<td>Proof of Closure</td>
<td>26</td>
</tr>
<tr>
<td>Proof of Convergence</td>
<td>29</td>
</tr>
<tr>
<td>Proof of our Algorithm if the System starts from a Correct State</td>
<td>50</td>
</tr>
<tr>
<td>Proof of our Algorithm if the System starts from an Incorrect state</td>
<td>53</td>
</tr>
<tr>
<td>Space Complexity of Our Algorithm</td>
<td>54</td>
</tr>
<tr>
<td>Message Complexity of Our Algorithm</td>
<td>54</td>
</tr>
<tr>
<td><strong>CHAPTER 5  CONCLUSION AND FUTURE WORK</strong></td>
<td>56</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>57</td>
</tr>
<tr>
<td>VITA</td>
<td>59</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I am indebted to my thesis advisory, Dr. Ajoy K. Datta, for chairing my committee and advising me throughout my thesis work. I owe a lot to Mrs.Doina Bein, Ph.D student UNLV, for her guidance and numerous contributions to this research. The support and advise offered by her for this thesis research was very valuable and important. I am also very grateful to Dr. Wolfgang Bein, Dr. Yoo-hwan Kim, and Dr. Venkatesan Muthukumar for their participation in my committee.

This thesis is dedicated to my parents for their love and constant encouragement. Last but not the least I would like to thank all my near and dear ones for their support.
CHAPTER 1

INTRODUCTION

1.1 Motivation

Finding better algorithms to sort a given set of data is an ongoing problem in the field of computer science. Sorting is informally defined as placing a given set of data in a particular order say, ascending or descending. It is the first step in solving a host of other algorithm problems. Indeed, "when in doubt, sort" is one of the first rules of algorithm design. A large effort is being put into the design and analysis of algorithms that are fully distributed. These algorithms are applicable in a network of nodes where no central controller is present, and no common clock is available. For these algorithms, a model commonly used contains a network of nodes, each with a unique identity known in the beginning only to it. Every node has only a local knowledge of the network, and its only means of communication is exchanging messages with its neighbors in the network. The messages arrive after a finite delay, but no a priori bound is known. We assume that a message contains a value, a control sequence, and the ID of the node to which it is intended to. It is usually assumed that any non-empty subset of the nodes starts the algorithm, and at the end each node has computed some function that is the result of that algorithm. In our solution to the sorting problem, a set of nodes which is privileged to move execute some receiving guard (if enabled). Once no more receiving guards are enabled at any process the values are sorted. Assuming that the computation cost and the queuing cost in each process are negligible compared to the communication cost, it is customary to measure the complexity of such algorithms by the total number of messages sent during any possible execution.
Self-stabilization is considered to be the most unified scheme to achieve fault
tolerance. We will study the problem of self-stabilizing sorting of \( n \) nodes in an
oriented chain.

### 1.2 Related Work

*Sorting* problems have been studied in sequential environment as well as parallel
one, due to their importance in computer science from both theoretical and practical
point of view. They however, have also attracted the attention of researchers in
distributed processing [3, 4, 5, 8, 12, 14]. There are various types of networks on
which sorting can be solved:

- A static storage of the network: sorting does not affect the network topology,
  just values are moved around. This is called in the literature as *static sorting with a
  reliable network*.

- A dynamic storage of the network: sorting may affect the network topology,
  where nodes may be moved around but values stick to their initial nodes. This is
called in the literature as *dynamic sorting with a reliable network*.

- A hybrid storage where both nodes and values may move around. This is called
  in the literature as *dynamic sorting with a unreliable network*.

Since in the literature there are self-stabilizing algorithms for maintaining the
network topology (so called local maintenance protocol), we solve the *distributed static sorting problem on a reliable network*. Many algorithms have been designed
and discussed for static distributed sorting with a reliable network. Besides solving
the sorting problem in a distributed fashion, finding a way to reduce the amount of
communication and time has become a problem for sorting as well. Gerstel et al [8]
proved the lower bound of the bit complexity for distributed sorting on a rooted tree
\( T \) with \( N \) nodes to be \( \Omega(\Delta_T \log \frac{L}{N}) \).

In [14] Zaks presented an algorithm for distributed ranking that uses, in the worst
case, at most \( O(n^2) \) messages. The algorithm is then extended to perform sorting,
which gives a worst case message complexity of $O(n^2)$ messages. Zaks algorithm is based on a tree structure. Messages from the nodes are sent to the root of the tree. Only the root node decides the rank of the key of $n$ nodes after receiving all messages. Thus the algorithm is not fully distributed one and neither a self-stabilizing one. The root may become a bottleneck of the computation.

In [5], a distributed sorting algorithm which is a variation of exchange sort i.e., neighboring elements that are out of order are exchanged, is presented. First, a sequential solution to the problem is presented which is subsequently transformed into a parallel solution. The transformation is triggered by the distribution of the data over nodes. The resulting algorithm has some flavor of odd-even transposition sort. They are, however, essentially different in two aspects. One difference is that a node does not communicate with only one of its two neighbors per iteration, but with both (as long as necessary). The other difference is that this algorithm is smooth, in the sense that the execution time is much less for almost sorted arrays than for hardly sorted arrays, with a smooth transition from one to the other behavior. Their algorithm which has a restricted local memory cannot solve a case where each node has exactly one element.

Luk and Ling [15] built a distributed sorting on a local area network. Their model is contrary to the conventional model that takes into account both the local processing time and the communication time. This model is intended to provide a framework within which the performance of various distributed sorting algorithms can be realistically analyzed. The distributed sorting algorithms are analyzed and implemented on Ethernet-connected Sun work stations. The empirical results by and large agree with the predictions derivable from the model.

A straight-line-topology local area network (LAN) to which a number of nodes are connected either in series or in parallel is considered in [11]. A file 'F' is arbitrarily partitioned among certain sites. The problem studied is that of rearranging the records of the file such that the keys of records at lower-ranking sites are all smaller
than those at higher-ranking sites. Lower bounds on the worst-case communication complexity are given for both the series and parallel arrangements, and algorithms optimal for all networks and files are presented by K.V.S. Ramarao.

Flocchini et. al. [10, 9] considered the problem of sorting on an anonymous ring network where all nodes are totally indistinguishable except for their input values. Initially, each vertex of the ring has associated a value from a totally ordered set, called multi-set. Besides considering the problem of sorting on a distributed multi-set they also investigate its relationship with the leader election problem.

Sasaki [13] has achieved a strict lower bound on the time complexity of $n - 1$ for distributed sorting on a synchronous line network where $n$ is the number of nodes. The lower time bound has traditionally been considered to be $n$ rounds because, in parallel sorting on a linear array, $n$ steps has been proven, based simply on the number of disjoint comparison-exchange operations. The strict optimal lower time bound of $n - 1$ is achieved by creating copies of elements. Despite the idea of creating nearly double the number of elements, the algorithm is faster than the odd-even transposition sort. This algorithm can be executed with both synchronous and asynchronous models by simply coping with wakeup. This $(n - 1)$ round algorithm can also have a great impact in the field of parallel algorithms. Sasaki designed a distributed sorting algorithm on an asynchronous line network [12] which is not only optimal on time complexity, but also on communication complexity. Moreover he has also shown its possible extensions. Our algorithm has some flavor of Sasaki’s [12] algorithm.

1.3 Contributions

In this thesis we will do research on designing a distributed sorting algorithm that is also self-stabilizing. We solve the distributed sorting problem in asynchronous systems, in case the values are not sorted, every process will send its maximum value to its right and its minimum value to its left neighbor in the network. Within
We give an overview of some topics involved in this research such as distributed sorting, synchronous and asynchronous systems, and self-stabilization in Chapter 2.

In Chapter 3 we present our self-stabilized distributed sorting algorithm, that includes the code and the data structures used to implement the algorithm.

The proof of correctness together with examples and complexities are given in Chapter 4.

We finish with concluding remarks and possible extensions of the algorithm in Chapter 5.
CHAPTER 2

PRELIMINARIES

In this chapter, we will present various terms used in our work. We define what a distributed system, synchronous and asynchronous systems are self-stabilization, oriented chains, lower and upper bounds on the time and space complexities. We also list few applications of sorting.

2.1 Distributed Systems

A distributed system is viewed as a collection of identical processing elements interconnected by a set of communication links in either regular or irregular pattern. Each processing element is an autonomous computer. These computers operate asynchronously, and communicate with each other by passing messages without central control. The term distributed system is used to describe communication networks, multiprocessor computers and a multitasking single computer. All the above variants of distributed systems have similar fundamental coordination requirements among the communicating entities, thus an abstract model that ignores the specific setting and captures the important characteristics of a distributed system is usually employed. A distributed system is modeled by a set of n state machines called nodes or process that communicate with each other. We usually denote the ith node in the system by Pi. Each node can communicate with the set of nodes, called neighbors. It is convenient to represent a distributed system by a node, and every two neighboring nodes are connected by a link of the communication graph. The communication can be carried out by message passing or by using a shared memory.
In the *message-passing* model, neighbors communicate each other by sending and receiving messages. In *shared-memory* model, nodes communicate by the use of shared communication registers. Nodes may write in a set of registers and may read from a possibly different set of registers. An atomic step is the largest step that is guaranteed to be executed uninterruptedly. Based on the granularity of the atomic steps we have two different models of shared memory model: *read-write* and *composite-atomicity* models. A node uses *composite-atomicity model* if each atomic step contains (at least) a read operation and a write operation. A node uses *read-write* atomicity model if some atomic step contains either a single read operation or a single write operation but not both.

Loui et. al. [7] had shown the non-existence of consensus protocol for systems that uses read-write model. The consensus protocol can be solved for systems that uses composite atomicity. Any system under composite atomicity model can be emulated by a system that uses read-write atomicity model. The composite atomicity model is strictly stronger than read-write atomicity model, and our algorithm will follow the composite atomicity model.

### 2.2 Synchronous and Asynchronous Systems

One of the most fundamental aspects of distributed systems is the distinction between *synchronous* and *asynchronous* systems. The asynchronous model of distributed systems has no bounds on:

- Execution latencies i.e., arbitrarily long (but finite) times may occur between execution steps of the same process.
- Message transmission latencies i.e., a message may be received an arbitrarily long time after the time it was sent.
- Clock drift rates i.e., process’s local clocks may be arbitrarily set.

In other words, the asynchronous distributed system model makes no assumptions about the time intervals involved in any behavior.
The synchronous model of distributed systems, conversely, has a priori known upper and lower bounds on these quantities:

- Each process has a bounded time between its execution steps.
- Each message is transmitted over a channel and received in a bounded time.
- Process's local clocks may drift either from each other or from global physical time only by a bounded rate.

The asynchronous model is the general case, and any proofs of its properties also apply to the synchronous model, but the converse is not true. However, a number of important properties (e.g. distributed agreement) have been proven impossible even under the weak conditions of asynchronous model, but are readily achievable in the synchronous model.

2.3 Self-stabilization

The concept of self-stabilization was introduced to computer science by Dijkstra, and was later authenticated by Lamport. The idea of self-stabilization was used in practice in other areas such as control theory, system science, etc even before Dijkstra had coined the name “self-stabilization”. There are various definitions available for self-stabilization, and unfortunately the researchers did not agree on one single definition. The widely accepted definition for self-stabilization with respect to behavior is given as follows: “A self-stabilizing system, regardless of its initial state, converges in finite time to a set of states that satisfy its specification”. And with respect to system state it is defined as “A self-stabilizing system, starting from an arbitrary state, reaches a state in finite time such that it starts behaving according to its specification”. The self-stabilization is defined in terms of two properties: Closure and Convergence. Closure refers to the property which requires that during all system executions, the system stays within some set of legal or desirable set of states unless a fault occurs. Convergence requires the system to reach a legal state from any arbitrary (possibly illegal) state in finite steps. A system is self-stabilizing
if it satisfies both closure and convergence properties. In [1, 2] a detailed study of different types of faults and how they are accommodated in their definition of stabilization (in terms of closure and convergence) was included. The term fault-tolerance is formally defined in [2] for the first time. It was shown that a fault-tolerant program is a composition of a fault-intolerant program and a set of fault-tolerant components. A method of designing multi-tolerant systems (a system that tolerate multiple types of faults) was also presented in [6].

Different models have been defined in the literature of self-stabilization. They are: execution model (message passing, and shared registers), fairness (strongly fair, unfair, weakly fair), granularity of an atomic step (composite and read/write atomicity), and types of daemons (central and distributed daemon). To prove that a system is self-stabilizing, two techniques have been commonly used. They are: convergence stair and variant function method. There are few general methods to design self-stabilization. We just list them without getting into the technical details: diffusing computation, silent stabilization, local stabilizer, local checking and local correction, window washing, self-containment, snap-stabilization, super-stabilization, power supply and transient faults.

There are numerous applications of self-stabilization in the area of network protocols, e.g routing, congestion control, sensor networks. Many self-stabilizing distributed solutions are proposed for graph theory problems, for example different types of spanning trees, maximal-matching, finding center and media, graph coloring. Some classical distributed algorithms are also solved using self-stabilization e.g., distributed reset, leader election, mutual exclusion, token circulation and termination detection. The above problems are solved for different topologies.

2.4 Oriented Chains

A line network is defined as a linear collection of \( n \) nodes \( P_1, P_2, \ldots, P_n \), where \( n > 2 \), and \( P_i \) is bidirectionally connected to \( P_{i+1} \), where \( 1 \leq i < n \). Without the loss
of generality, we assume that the network is laid horizontally such that the process $P_i$, also called \textit{LEFT}, is at the endpoint on the left. Furthermore, we assume that each process knows its neighbors only by the local names "left" and "right", with the orientation consistent along the line. Note that an endpoint process knows that it is an endpoint by the fact that one of the corresponding local names is "null". Moreover we assume that any process $P_i$ has no knowledge of its position $i$ and the total number of nodes $n$. If a process $P_i$ has $k$ elements to be sorted, then each process requires a local memory of at least $O(k \log L)$ bits.

2.5 Lower and Upper bounds

Lower bound is informally defined as a function or growth rate below which solving a problem is impossible. An asymptotic lower bound is defined as a function of the size of the input, on the best (fastest, least amount of space used, etc.) an algorithm can possibly achieve to solve a problem. That is, no algorithm can use fewer resources than the bound. An asymptotic upper bound is defined, as function of the size of the input, on the worst case (slowest, most amount of space used, etc.) an algorithm will do to solve a problem. That is, no input will cause the algorithm to use more resources than the bound.

In [7] of Loui, the message complexity of sorting problem for ring network was studied, where initial values have to be sorted clockwise, starting at any position. It was shown that every sorting algorithm requires $\Omega(N^2 \log (L/N))$ bits on a ring of size $N$ (where $L$ is the maximum number of bits to store a value), and an algorithm was presented that achieves this lower bound. Similar results are shown for meshes. Sasaki has achieved a lower bound of $n - 1$ rounds in a non-stabilizing system. O'Grestel and Zaks have obtained $\Delta_T + \log (L/N)$ bits in the worst case for a sorting problem with tree topology, where $\Delta_T$ is the maximum degree of a node in tree $T$. 

10

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
2.6 Applications of Sorting

Sorting is used as the basic building block, because once items are sorted many other problems becomes easy. We will list few applications of sorting: searching, closest pair, element uniqueness, frequency distribution, selection etc.
CHAPTER 3

ALGORITHM AND DATA STRUCTURES

In this chapter, we will present the variables, the predicates, the macros used by our distributed program executed in some node \( i \) and the code. The distributed program is semi-uniform (i.e., except for nodes LEFT and RIGHT that execute a different program and all the other nodes in the network have the same program).

3.1 Variables

The variables of a node are either link-shared, shared or local variables. The link-shared variables are "attached" to the link: some adjacent process can either read or write only in that variable. The shared variables are readable by both neighbors of the process. The local variables are not accessible by other nodes.

Each node \( i \) will be maintaining the following local variables, consistent with the rest of the linear network. Variable

- \( \text{left}.i \) indicates the ID of the left neighbor of node \( i \).
- \( \text{right}.i \) indicates the ID of the left neighbor of node \( i \).
- \( U_i \) is the initial element that each process holds.
- \( Rcvd \) is a set of elements initially \( \{u_i\} \).

Each node \( i \) will be maintaining four link-shared variables \( M_{ir}, M_{is}, M_{rr}, M_{rs} \).

Variable

- \( M_{is} \) is used to store the values that are to be sent to the left neighbor of the node \( i \) (\( \text{left}.i \)).
- \( M_{ir} \) is used to store the values received from the left neighbor of node \( i \).
• $M_{rs}$ is used to store the values that are to be sent to the right neighbor of node $i$ ($right.i$).

• $M_{rr}$ is used to store the values received from the right neighbor of node $i$.

The variables $M_{lr}, M_{ls}, M_{rr}, M_{rs}$ have the field:

• $val$ which is either $u_L$ if a node $i$'s state is right or $u_R$ if a node $i$'s state is left.

Each node $i$ will be maintaining the following shared variables. Variable

• $state$ is used to indicate one of the five possible states: $state \in \{left, receive, relay, right, send\}$.

• $v_L, v_R$ are the received elements from left and right neighbors correspondingly.

• $u_L, u_R$ are the elements sent to the left and right neighbors correspondingly.

![Diagram of variables used in our algorithm](image)

* denotes the link state shared variable

Figure 1. Diagram to show the variables used in our algorithm

3.2 Predicates

The predicates used in the algorithm are defined as follows:
•  **to\_send\_min(i)**: When a node has one or more values and if its values are less than the values of its left neighbor's value then this predicate will be enabled, and the node sends its minimum value to its left neighbor.

•  **to\_send\_max(i)**: When a node has one or more values and if its values are greater than the values of its right neighbor's value then this predicate will be enabled, and sends its maximum value to its right neighbor.

•  **to\_rcv\_right\_msg(i)**: Whenever a node's right neighbor has a new value to send this predicate is enabled.

•  **to\_rcv\_left\_msg(i)**: Whenever a node's left neighbor has a new value to send this predicate is enabled.

•  **enable\_send\_right**: If a node's right neighbor ∈ \{receive, right, relay, send\} and if the node's to\_send\_max predicate is true then Predicate enable\_send\_right becomes true.

•  **enable\_send\_left**: If a node's left neighbor ∈ \{receive, left, relay, send\} and if the node's to\_send\_min predicate is true then Predicate enable\_send\_left becomes true.

**Predicates:**
\[
\begin{align*}
to\_send\_min(i) & \equiv Rcvd.i \neq \emptyset \land max(Rcvd.left(i)) > min(Rcvd.i) \\
to\_send\_max(i) & \equiv Rcvd.i \neq \emptyset \land min(Rcvd.right(i)) < max(Rcvd.i) \lor |Rcvd.i| > 1 \\
to\_rcv\_right\_msg(i) & \equiv Mr. ,i \neq Mr.right(i) \\
to\_rcv\_left\_msg(i) & \equiv Mr.i \neq Mr.left(i) \\
enable\_send\_right(i) & \equiv (to\_send\_max(i) \land state.right(i) \in \{receive, right, relay, send\}) \\
enable\_send\_left(i) & \equiv (to\_send\_min(i) \land state.left(i) \in \{receive, left, relay, send\})
\end{align*}
\]

3.3 The Macros

There are a number of macros used by the algorithm.

\[
\text{send\_Min}(i) = \text{process } i \text{ sends its minimum element to its left neighbor.}
\]
$u_{L.i} := \min(Rcvd.i)$

$M_{ls.i}.val := u_{L.i}$

$Rcvd.i := Rcvd.i \setminus \{u_{L.i}\}$

$send_{Max}(i) =$ process $i$ sends the maximum element to its right neighbor.

$u_{R.i} := \max(Rcvd.i)$

$M_{rs.i}.val := u_{R.i}$

$Rcvd.i = Rcvd.i \setminus \{u_{R.i}\}$

$rcvd_{right}(i) =$ process $i$ receives the minimum element of its right neighbor.

$M_{rr.i} = M_{ls.right.i}$

$v_{R.i} = M_{ls.right.i}.val$

$Rcvd.i = Rcvd.i \cup \{v_{R.i}\}$

$rcvd_{left}(i) =$ process $i$ receives the maximum element of its left neighbor.

$M_{lr.i} = M_{rs.right.i}$

$v_{L.i} = M_{rs.right.i}.val$

$Rcvd.i = Rcvd.i \cup \{v_{L.i}\}$
3.4 The Code

Let LEFT be the leftmost process of the linear network, and RIGHT be the rightmost process of the linear network. By abuse of notation, let all other nodes be called MIDDLE nodes. The code for the end-nodes is different from the code of the middle-nodes.

Algorithm 1 Stabilizing Sorting on Linear Networks

Actions for LEFT node $i$ such that $\text{left}_i = \perp$:

$S :: \text{state}_i = \text{send} \land \text{enable}_\text{send}_\text{right}(i) \rightarrow$

if $\text{Rcvd}_i \neq \emptyset$ then $\text{send}_\text{Max}(i)$

$\text{state}_i = \text{receive}$

$SR :: \text{state}_i = \text{send} \land \neg\text{to}_\text{send}_\text{max}(i) \land \text{to}_\text{rcv}_\text{right}_\text{msg}(i) \rightarrow \text{state}_i = \text{receive}$

$R.R :: \text{state}_i = \text{receive} \land \text{rcv}_\text{right}(i)$

$\text{state}_i = \text{relay}$

$R.S :: \text{state}_i = \text{relay} \land \text{state}_\text{right}(i) = \text{receive} \rightarrow \text{state}_i = \text{send}$

$R.R :: \text{state}_i = \text{relay} \land \text{state}_\text{right}(i) = \text{send} \rightarrow \text{state}_i = \text{receive}$

$RS :: \text{state}_i = \text{receive} \land (\neg\text{to}_\text{rcv}_\text{right}_\text{msg}(i) \land \text{to}_\text{send}_\text{max} \land \text{state}_\text{right}(i) \neq \text{send})$

$\rightarrow \text{state}_i = \text{send}$

$R.S :: \text{state}_i = \text{relay} \land \text{state}_\text{right}(i) = \text{relay} \land \text{to}_\text{send}_\text{max} \rightarrow \text{state}_i = \text{send}$

$L.S :: \text{state}_i = \text{left} \rightarrow \text{state}_i = \text{send}$

$R.S :: \text{state}_i = \text{right} \rightarrow \text{state}_i = \text{send}$

16

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Actions for RIGHT node \( i \) such that \( \text{right}.i = \perp \):

\[ S :: \text{state}.i = \text{send} \land \text{enable.send}.\text{left}(i) \rightarrow \\
\quad \text{if} \ \text{Rcvd}.i \neq \emptyset \ \text{then} \ \text{send.Min}(i) \\
\quad \text{state}.i = \text{receive} \]

\[ SR :: \text{state}.i = \text{send} \land ((\neg \text{to.send}.\text{min}(i) \land \text{to.rcv}.\text{left}.\text{msg}(i)) \lor \text{state}.\text{left}(i) = \text{send}) \rightarrow \\
\quad \text{state}.i = \text{receive} \]

\[ R_\text{Re} :: \text{state}.i = \text{receive} \land \text{to.rcv}.\text{left}.\text{msg}(i) \rightarrow \\
\quad \text{rcvd}.\text{left}(i) \\
\quad \text{state}.i = \text{relay} \]

\[ R_\text{e.S} :: \text{state}.i = \text{relay} \land \text{state}.\text{left}(i) = \text{receive} \rightarrow \text{state}.i = \text{send} \]

\[ R_\text{e.R} :: \text{state}.i = \text{relay} \land \text{state}.\text{left}(i) = \text{send} \rightarrow \text{state}.i = \text{receive} \]

\[ R_\text{S} :: \text{state}.i = \text{receive} \land ((\neg \text{to.rcv}.\text{left}.\text{msg}(i) \land \text{to.send}.\text{min}) \land \text{state}.\text{left}(i) \neq \text{send}) \rightarrow \\
\quad \text{state}.i = \text{send} \]

\[ R_\text{eS} :: \text{state}.i = \text{relay} \land \text{state}.\text{left}(i) = \text{relay} \land \text{to.send}.\text{min} \rightarrow \text{state}.i = \text{send} \]

\[ L_\text{S} :: \text{state}.i = \text{left} \rightarrow \text{state}.i = \text{send} \]

\[ R_\text{i.S} :: \text{state}.i = \text{right} \rightarrow \text{state}.i = \text{send} \]
Actions for MIDDLE node i:

\[ S :: \text{state}.i = \text{send} \land (\text{enable}_{\text{send}.right} \lor \text{enable}_{\text{send}.left}) \rightarrow \]
\[ \begin{align*}
& \text{if}(\text{Rcvd}.i \neq \emptyset \land \text{enable}_{\text{send}.left}) \text{ then send}_{\text{Min}}(i) \\
& \text{if}(\text{Rcvd}.i \neq \emptyset \land \text{enable}_{\text{send}.right}) \text{ then send}_{\text{Max}}(i) \\
& \text{state}.i = \text{receive}
\end{align*} \]

\[ SR :: \text{state}.i = \text{send} \land ((\neg \text{to}_{\text{send}.min}(i) \land \neg \text{to}_{\text{send}.max}(i)) \lor \text{state}.left(i) = \text{send}) \rightarrow \text{state}.i = \text{receive} \]

\[ \mathcal{R} . \text{Re} :: \text{state}.i = \text{receive} \land \text{to}_{\text{rcv}.right}.msg(i) \land \text{to}_{\text{rcv}.left}.msg(i) \rightarrow \]
\[ \begin{align*}
& \text{rcvd}.right(i) \\
& \text{rcvd}.left(i) \\
& \text{state}.i = \text{relay}
\end{align*} \]

\[ \mathcal{R} . \text{Ri} :: \text{state}.i = \text{receive} \land \text{to}_{\text{rcv}.right}.msg(i) \land \neg \text{to}_{\text{rcv}.left}.msg(i) \rightarrow \]
\[ \begin{align*}
& \text{rcvd}.right(i) \\
& \text{state}.i = \text{right}
\end{align*} \]

\[ \mathcal{R} . \text{L} :: \text{state}.i = \text{receive} \land \neg \text{to}_{\text{rcv}.right}.msg(i) \land \text{to}_{\text{rcv}.left}.msg(i) \rightarrow \]
\[ \begin{align*}
& \text{rcvd}.left(i) \\
& \text{state}.i = \text{left}
\end{align*} \]

\[ \mathcal{R} S :: \text{state}.i = \text{receive} \land (\neg \text{to}_{\text{rcv}.right}.msg(i) \land \neg \text{to}_{\text{rcv}.left}.msg(i)) \land \]
\[ ((\text{to}_{\text{send}.max}(i) \land \text{state}.right(i) \neq \text{send}) \lor (\text{to}_{\text{send}.min}(i) \land \text{state}.left(i) \neq \text{send})) \rightarrow \text{state}.i = \text{send} \]

\[ \mathcal{R} . \text{Re} :: \text{state}.i = \text{right} \land \text{to}_{\text{rcv}.left}.msg(i) \rightarrow \]
\[ \begin{align*}
& \text{rcvd}.left(i) \\
& \text{state}.i = \text{relay}
\end{align*} \]

\[ \mathcal{L} . \text{Re} :: \text{state}.i = \text{left} \land \text{to}_{\text{rcv}.right}.msg(i) \rightarrow \]
\[ \begin{align*}
& \text{rcvd}.right(i) \\
& \text{state}.i = \text{relay}
\end{align*} \]
Actions for MIDDLE node $i$ (continued):

$L.S :: state.i = left \land \neg to.rcv.right.msg(i) \rightarrow state.i = send$

$L.R :: state.i = left \land to.rcv.left.msg(i) \rightarrow state.i = receive$

$R.S :: state.i = right \land \neg to.rcv.left.msg(i) \rightarrow state.i = send$

$R.R :: state.i = right \land to.rcv.right.msg(i) \rightarrow state.i = receive$

$Re.S1 :: state.i = relay \land state.left(i) = send \land state.right(i) = send \rightarrow state.i = receive$

$Re.S2 :: state.i = relay \land state.left(i) = send \land state.right(i) = receive \rightarrow$

\hspace{2em} state.i = receive

$Re.R1 :: state.i = relay \land state.left(i) = receive \land state.right(i) = receive \rightarrow$

\hspace{2em} state.i = send

$Re.R2 :: state.i = relay \land state.left(i) = receive \land state.right(i) = send \rightarrow$

\hspace{2em} state.i = receive
3.5 Explanation of Our Algorithm

Model: A distributed system is an undirected, connected graph $S = (V, E)$ where $V$ is the set of nodes ($|V| = n$) and $E$ is the set of links or edges. A link between nodes $i$ and $j$ is identified by the pair $(i, j)$ and for every $(i, j) \in E$, nodes $i$ and $j$ are called neighbors. We consider the end nodes of a line network to be special nodes i.e., the leftmost process to be LEFT and the rightmost process to be RIGHT, and the intermediate nodes to be MIDDLE nodes. Each process/node will have a set of elements to be sorted. Each node executes the program which is specified by a finite set of variables and finite set of actions. Each node execute asynchronously.

Communications: Each node has three types of variables: LinkShared variables, Shared variables and Local variables. More details of these variables are given in the variables section. 3.1. The shared and the link-shared variables are used to communicate with the neighbors. The local variables defined in the program of the node $i$ are used strict locally, meaning that they cannot be accessed by the neighbors of $i$. A node can only write to its own variables and can only read variables owned by the neighboring nodes. So, only the shared and the link shared variables of $i$ can be accessed by $i$ and its neighbors.

States and Configurations: The state of node is defined by the values of its local variables. A configuration of a distributed system $S = (V, E)$ is an instance of the states of its nodes and links. The set of configurations of $S$ is denoted as $C$.

Actions and Computation: A nodes action consists of an internal computation along with one or more read and write actions. This execution model is called composite atomicity model. Each action is defined by a labeled guarded command:

\[ <\text{label}> :: <\text{guard}> \rightarrow <\text{statement}> \]

The guard of an action in the program of $i$ is a boolean expression involving the local variables of $i$, the link shared variables, shared variables of $i$ and its neighbors. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the atomic execution of an action of $i$ is called
a step of \( i \). A node's action may change the global configuration of the system. Furthermore, several actions may occur at the same time. We define the space complexity of a self-stabilizing protocol as the memory space needed to hold the local, link shared and the shared variables. We also define the time complexity of a self-stabilizing protocol as the time needed to reach a configuration that matches the global predicate \( P_{\text{SORT}} \) (defined in next chapter) after the faults cease to occur.

Basic idea of our algorithm: The leftmost node or process (i.e., LEFT or \( P_1 \)) will send its maximum element to its right neighbor if it has any element and that element is not sorted. In the same way, a rightmost node or process (i.e., RIGHT or \( P_n \)) will send its minimum element to its left neighbor if it has any element and that element is not sorted. Each other node or process (i.e., MIDDLE or \( P_2, \ldots, P_{n-1} \)) will send its maximum element to its right neighbor if the right neighbor has some value which is lesser than its maximum element. It also sends its minimum element to its left neighbor if the left neighbor has some value which is larger than its minimum element. This transfer of elements continues until all the elements in the linear network are in sorted order.

Explanation of our algorithm (informal description of the algorithm): Each process or node has a set of elements in its local variable called \( \text{Rcvd} \), which holds the elements for sorting. Each process has a local variable called \( \text{state} \) which can be in one of the five values \{left, receive, relay, right, relay\}. As previously mentioned we divide the linear network into three groups LEFT, RIGHT, MIDDLE. We now show the operations of the node or process with respect to their variable \( \text{state} \).

When \( P_i \)'s variable \( \text{state} \) is left or right: If the leftmost node or process (LEFT or \( P_1 \)) variable \( \text{state} \) is left or right then no messages will be received or sent, the node just changes its variable \( \text{state} \) to send.

In the same way if the rightmost node's (RIGHT or \( P_n \)) variable \( \text{state} \) is left or right then the RIGHT node will not send or receive messages but just changes its variable \( \text{state} \) to send.
If the MIDDLE node's variable state is left, and if it has to receive messages from its right, it receives the message and changes its variable state to relay. If it has to no messages from its right neighbor, it changes its state variable to send. If it has no messages from it left neighbor, it simply changes its variable state to receive. If the MIDDLE node's variable state is right and if it has any message from its left neighbor then it receives the message and changes its variable state to relay. If the MIDDLE node has any message from its right neighbor then it changes its variable state to receive. If it has no messages from its left neighbor then the MIDDLE node's variable state is changed to send.

When \( P_i \)'s variable state is send: If the leftmost nodes variable state is send and if it has some element which is larger than its neighboring node and if its neighboring nodes variable state is either receive, right, relay, or send it sends its maximum element and changes its variable state to receive. If the leftmost node has no element which is larger than its neighboring node or if the variable Rcvd of the leftmost node is empty then the leftmost nodes variable state is changed to receive.

If the rightmost nodes variable state is send and if it has some element which is smaller than its neighboring node and if its neighboring nodes variable state is either receive, left or relay, it sends its minimum element and changes its variable state to receive. If the rightmost node has no element which is smaller than its neighboring node or if the variable Rcvd of the rightmost node is empty then the leftmost nodes variable state is changed to receive.

If the MIDDLE node's variable state is send and if it has some element which is larger than its right neighbor or it has some element which is smaller than its left neighbor, it sends them to its right and left neighbors respectively, and changes its variable state to receive. If the MIDDLE node has no elements to send or if the MIDDLE nodes left neighbor's variable state is send then the MIDDLE node remains unchanged without receiving or sending any messages.

When \( P_i \)'s variable state is receive: If the leftmost node's (LEFT or \( P_i \)) variable
state is receive and if it has received messages from its right neighbor then the variable state of the LEFT node is changed to relay. If the leftmost node has no messages from its right neighbor, and it has to send its maximum value to its right, then it checks if the variable state of its right neighbor is not send. If so then the variable state of the LEFT node is changed to send.

Similarly for the rightmost node (RIGHT or $P_n$) if the variable state is receive and it has messages from its left then the variable state of the RIGHT node is changed to relay after receiving that left message. If the rightmost node has no message from its left neighbor but it has to send its minimum element to its left, it checks whether the left neighbor's variable state is not send, if so the rightmost state variable is changed to send.

If the MIDDLE nodes variable state is receive and it has both left and right messages, it receives them and changes its variable state to relay. If the MIDDLE node has either right or left messages, then it receives those messages and changes its variable state to right and left respectively. If the MIDDLE node has no left or right messages, and if it has to send either its maximum or minimum element and the variable state of the left or right neighbor is not send, then the variable state of the MIDDLE node is changed to send.

When $P_l$'s variable state is relay: If the leftmost node's variable state is relay, it checks whether its right neighbors variable state is either receive, send or relay. If its right neighbor's variable state is receive then the leftmost node changes its variable state to send without sending or receiving messages. If its right neighbors variable state is send then the variable state of the leftmost node changes to receive without sending or receiving messages. If its right neighbor's variable state is relay, and if the middle node has to send its maximum value to its right neighbor then the variable state of the middle node is changed to send.

Similarly if the rightmost node's variable state is relay, it checks whether its left neighbor's variable state is either receive, send or relay. If its left neighbor's
variable state is receive then the variable state of the leftmost node is simply changed to send. If its left neighbor's variable state is send then the variable state of the leftmost node is changed to receive and if the left neighbors variable state is relay and the middle node has to send its minimum value then the variable state of the middle node is changed to send.

If the MIDDLE node's variable state is relay, it checks the variable state of its left and right neighbors. If the variable state of its left neighbor is send and if the variable state of its right neighbor is either send or receive then the MIDDLE node just changes its value of the variable state to receive. If the left neighbor's variable state is receive and its right neighbors variable state is also receive then the MIDDLE node changes its variable state to send without receiving or sending any message. If the left neighbor's variable state is receive and right neighbors variable state is send the MIDDLE node changes its variable state to receive without sending or receiving any messages.
CHAPTER 4

PROOF OF CORRECTNESS

We define the Global Predicate $P$ as follows:

$P_{\text{SORT}} = L_1 \land L_2 \land L_3$ where

- $L_1 = \{ \forall \text{node } i : \text{left}(i) \neq \perp \land \max(Rcvd.left(i)) \leq \min(Rcvd.i) \}$
- $L_2 = \{ \text{no new messages are generated} \}$
- $L_3 = \exists k \geq 0 : (\forall i, 1 \leq i < k : |Rcvd.i| = 1) \land (\text{if } \exists i, k \leq i \leq n \text{ then } |Rcvd.i| = 0)$

Algorithm $S$ is self-stabilizing if it satisfies the following two conditions:

A (Convergence) Starting from some arbitrary configuration $C_0$, in finite number of rounds the algorithm reaches a configuration that satisfies Predicate $P$.

B (Closure) If a configuration $C$ satisfies Predicate $P$, all subsequent configurations $C'$ reachable from $C$ must satisfy Predicate $P$.

We prove that Algorithm $S$ satisfies the above two condition.

4.1 Proof of Closure

Property 1. If some node $i$ is in a configuration that satisfies Predicate $P$ then its variable $Rcvd.i$ remains unchanged and node $i$ does not generate any messages.
Proof. Since Predicate $L_1$ is true in configuration $C$, $\forall$ node $i$, the predicates $to\_send\_max(i)$ and $to\_send\_min(i)$ are false. Thus Guard $S$ is disabled for any node $i$, so macros $send\_Min(i)$ and $send\_Max(i)$ do not get executed. Thus variable $Rcvd(i)$ remains unchanged. Since Predicate $L_2$ is true in configuration $C$, $\forall$ node $i$, the predicates $to\_rcv\_right\_msg(i)$ and $to\_rcv\_left\_msg(i)$ are false. We have three cases, based on node $i$ position:

a) $i = LEFT$ and the value of its variable $state$ is

$send ::$ Guard $SR$ is disabled. No action is enabled and so no new messages are generated.

receive :: Guard $R$ is disabled. No new messages are generated and variable $Rcvd(i)$ remains unchanged.

relay :: Depending upon the value of $state.right(i)$, a node $i$ may change its variable $state$ to $send$, $receive$ or remain in $relay$. In any case, no new messages are generated.

b) $i = RIGHT$ and the value of its variable $state$ is

$send ::$ If the value $state.left(i) = send$ then Guard $SR$ is enabled. Node $i$ changes its state to $receive$. No new messages are generated.

receive :: Guard $R$ is disabled. No new messages are generated and variable $Rcvd(i)$ remains unchanged.

relay :: Depending upon the value of $state.left(i)$, one of the guards $Re\_S$ or $Re\_R$ may be enabled. Node $i$ may change its variable $state$ to $send$, $receive$ or remain in $relay$ without generating any new messages.

c) $i$ is a MIDDLE node and the value of its variable $state$ is
send :: Depending upon the value of state.left(i), Guard SR may be enabled. Node i may change its variable state to receive, without generating any new messages.

receive :: Guards R.Re, R.Ri, R.L are disabled. No action is enabled and so no new messages are generated and also variable Rcvd(i) remains unchanged.

relay :: Depending upon the values of state.left(i) and state.right(i), node i may change its variable state to send, receive or remain in relay, without generating any new messages.

left :: Guards L.Re and L.R are disabled and variable Rcvd(i) remains unchanged. Guard L.S is enabled. Node i changes its variable state to send, without generating any new messages.

right :: Guards R.i.Re and R.R are disabled and variable Rcvd(i) remains unchanged. Guard R.S is enabled. Node i changes its variable state to send, without generating any new messages.

Lemma 1. (Closure) If a configuration C satisfies Predicate P, all subsequent configurations C' reachable from C satisfy Predicate P.

Proof. Configuration C satisfies Predicate P. Starting from configuration C, if some node i executes and in one execution step reaches configuration C', then we show that the subsequent configuration C' does also satisfy Predicate P. (We show that configuration C' satisfies L1, L2 and L3.)

In other words, we prove in Property 1 that: - Node i does not generate any new messages and does not receive messages. (Predicate L2 is satisfied.) - Variable Rcvd.i remains unchanged. (Predicates L1 and L3 are satisfied.)

4.2 Proof of Convergence

27
**Property 2.** If \( \text{state.LEFT} = \text{send} \) and \( \text{state.right(LEFT)} = \text{left} \) then in finite number of rounds the variable \( \text{state.right(LEFT)} \) will have one of the values in the set \{receive, right, relay\}.

**Proof.** If the variable \( \text{state.right(LEFT)} = \text{left} \), then we have the following sub cases:

1) If Predicate \( \text{to.recv.right.msg(right(LEFT))} \) is true, Guard \( LRe \) is executed and it changes the variable \( \text{state.right(LEFT)} \) to relay in at most one round.

2) If Predicate \( \text{to.recv.right.msg(right(LEFT))} \) is false, Guard \( L\text{S} \) is executed and it changes the variable \( \text{state.right(LEFT)} \) to send in at most one round. Once that variable \( \text{state.right(LEFT)} \) variable is send, Guard \( SR \) is enabled. It is also possible that Guard \( S.R \) may be enabled. In any case, one of the enabled guards will be selected and executed in at most one round, so the variable \( \text{state.right(LEFT)} \) becomes receive. So a total of at most two rounds are needed for the variable \( \text{state.right(LEFT)} \) to have a value in the set \{receive, right, relay\}, and here it is receive.

3) If Predicate \( \text{to.recv.left.msg(right(LEFT))} \) is true then Guard \( L\text{R} \) is the only enabled guard at node \( \text{right(LEFT)} \). It is selected and executed in at most one round, and it changes the variable \( \text{state.right(LEFT)} \) to receive.

\( \square \)

**Lemma 2.** If \( |Rcvd.LEFT| > 1 \), in finite number of steps variable \( \text{Rcvd.LEFT} \) has only one value \( |Rcvd.LEFT| = 1 \).

**Proof.** If \( |Rcvd.LEFT| > 1 \), then Predicate \( \text{to.send.max(LEFT)} \) is true.

The variable \( \text{state.LEFT} \) can take values in the set \{send, receive, relay, left, right\}.

We have several cases:

**Case 1) state.LEFT = send.** We have several sub cases, depending on the value of \( \text{state.right(LEFT)} \).
1.i) If the variable $\text{state.right}(i) \in \{\text{right, receive, relay, send}\}$ then Guard $S$ is enabled at node $LEFT$ and no other guards are enabled at node $LEFT$. Node $LEFT$ will send the maximum value of $Rcvd.LEFT$ to its right neighbor in at most one round. Thus $|Rcvd.LEFT|$ decreases by 1.

1.ii) If the variable $\text{state.right}(LEFT) = \text{left}$, then in finite amount of rounds (at most two rounds) Guards $S$ or $SR$ will be enabled (Property 2). One of the enabled guards is arbitrarily selected and will change the variable $\text{state.right}(LEFT)$ to $\text{receive}$. Then we apply Case 1.i).

*Case 2*) $\text{state.LEFT} = \text{relay}$. Depending upon the $\text{state.right}(LEFT)$ we have several cases:

2.i) If the variable $\text{state.right}(LEFT) = \text{receive}$ then Guard $Re.S$ will be enabled at node $LEFT$ and it is the only enabled guard. It will be executed in at most one round and $\text{state.LEFT}$ becomes $\text{send}$.

2.ii) If the variable $\text{state.right}(LEFT) = \text{send}$ then Guard $Re.R$ will be enabled at node $LEFT$ and it is the only enabled guard. It will be executed in at most one round and $\text{state.LEFT}$ becomes $\text{receive}$.

2.iii) If the variable $\text{state.right}(LEFT) = \text{relay}$ then Guard $ReS$ will be enabled at node $LEFT$ and it is the only enabled guard. It will be executed in at most one round and $\text{state.LEFT}$ becomes $\text{send}$.

2.iv) If the variable $\text{state.right}(LEFT) = \text{left}$ then at node $\text{right}(LEFT)$ one of the Guards $L.R$, $L.S$, or $L.Re$ is enabled and one guard will be selected and executed in at most one round. The variable $\text{state.right}(LEFT)$ is changed to $\text{relay}$, $\text{send}$ or $\text{receive}$. After this execution, if the node $LEFT$ is still in $\text{relay}$ then it changes its variable $\text{state}$ by following either Case 2.ii) or Case 2.iii) in one more round. So a total of at most three rounds are necessary.
2.v) If the variable \( \text{state.right(LEFT)} = \text{right} \) then at node \( \text{right(LEFT)} \) one of the Guards \( \text{Ri.Re, RS, or R.R} \) is enabled and one guard will be selected and executed in at most one round. The variable \( \text{state.right(LEFT)} \) is changed to \( \text{relay, send or receive} \) in at most one round. After this execution, if the node \( \text{LEFT} \) is still in \( \text{relay} \) then it changes its variable \( \text{state} \) by following either Case 2.ii) or Case 2.iii) in at most one more round. So a total of three rounds are necessary.

**Case 3)** \( \text{state.LEFT} = \text{receive} \). Then either Guard \( \text{R} \) or \( \text{RS} \) is enabled at node \( \text{LEFT} \) (but one of them is certainly enabled and no other guard is enabled).

3.i) If Predicate \( \text{to.rcv.right.msg(LEFT)} \) is \( \text{true} \), then Guard \( \text{R} \) is enabled. In at most one round, node \( \text{LEFT} \) changes value of the variable \( \text{state.LEFT} \) to \( \text{relay} \). Then Case 2) is applied.

3.ii) If Predicate \( \text{to.rcv.right.msg(LEFT)} \) is \( \text{false} \) then Guard \( \text{RS} \) is enabled. In at most one round, node \( \text{LEFT} \) changes value of the variable \( \text{state.LEFT} \) to \( \text{send} \). Then Case 1) is applied.

**Case 4)** \( \text{state.LEFT} = \text{left} \). Then at node \( \text{LEFT} \) only Guard \( \text{LS} \) is enabled. It will be selected and executed in at most one round, and the variable \( \text{state.LEFT} \) changes to \( \text{send} \). Then Case 1) will be applied. **Case 5)** \( \text{state.LEFT} = \text{right} \). Then at node \( \text{LEFT} \) only Guard \( \text{Ri.S} \) is enabled. It will be selected and executed in at most one round, and the variable \( \text{state.LEFT} \) changes to \( \text{send} \). Then Case 1) will be applied.

**Lemma 3.** If some value in \( \text{Rcvd.LEFT} \) is greater than some value in \( \text{Rcvd.right(LEFT)} \) (i.e., \( \max(\text{Rcvd.LEFT}) > \min(\text{Rcvd.right(LEFT)}) \)) then the value will be eventually moved from \( \text{Rcvd.LEFT} \) to \( \text{Rcvd.right(LEFT)} \).

**Proof.** Let \( x \) be the maximum in \( \text{Rcvd.LEFT} \) (i.e. \( x \geq \{\text{Rcvd.LEFT}\ \backslash \{x\}\} \)) and \( y \) be the minimum in \( \text{Rcvd.right(LEFT)} \) (i.e. \( y \leq \{\text{Rcvd.right(LEFT)} \ \backslash \{y\}\} \)). We have...
that $x > y$. Predicate $\text{to\_send\_max(LEFT)}$ is true at node $LEFT$ and Predicate $\text{to\_send\_min(right(LEFT))}$ is true at node $right(LEFT)$. Then we follow the same proof as Property 2 to show that $x$ is removed from $Rcvd.LEFT$ (Macro $send._Max(LEFT)$ is executed at node $LEFT$) in finite number of rounds (at most three rounds). The value which may be sent by node $right.LEFT$ is $y$ and since $y \leq \{Rcvd.right(LEFT) \setminus y\}$ and $y < x$, it implies that by adding $x$ to $Rcvd.right(LEFT) \setminus y$ we converge to the situation when $\max(Rcvd.LEFT) \leq \min(Rcvd.right(LEFT))$.

**Property 3.** If for the node $i = LEFT$ the condition: 

\[ |Rcvd.i| = 1 \text{ and any value in } Rcvd.i \text{ is less than or equal to any value in } Rcvd.right(i) \]: $|Rcvd.i| = 1 | \land \max(Rcvd.i) \leq \min(Rcvd.right(i))$, then no new messages are generated by node $LEFT$.

**Proof.** If the above condition is true, then Predicate $\text{to\_send\_max(LEFT)}$ is false, thus Guard $S$ is disabled at node $LEFT$. Guard $S$ is the only guard whose execution can generate new messages from node $LEFT$.

**Property 4.** If the variables $\text{state.RIGHT} = send$ and $\text{state.left(RIGHT)} = right$ then in finite number of rounds the variable $\text{state.left(RIGHT)}$ will have one of the values in the set \{left, receive, relay\}.

**Proof.** If the variable $\text{state.left(RIGHT)} = right$, then we have the following sub cases:

1) If Predicate $\text{to\_rcv\_left\_msg(left(RIGHT))}$ is true, then Guard $Ri.Re$ is enabled. Guard $R.R$ may be enabled as well. One of the enabled guards is selected and executed in at most one round at node $left(RIGHT)$. Variable $\text{state.left(RIGHT)}$ becomes either $relay$ or $receive$ (and then Case 1.i) of Property 5 is applied).

31
2) If Predicate $\text{to}_r \text{c}_v \text{.left}_m \text{sg}(\text{left}(\text{RIGHT}))$ is false, Guard $R.S$ is enabled and also Guard $R.R$ may be enabled as well. One of the enabled guards is selected and executed in at most one round at node $\text{left}(\text{RIGHT})$. Variable $\text{state}_l \text{eft}(\text{RIGHT})$ becomes either $\text{send}$ (Case a) is applied or $\text{receive}$ (and then Case 1.i) of Property 5 is applied).

Lemma 4. If Predicate $\text{to}_s \text{end}_m \text{in}_\text{i}_\text{m}$ is true at node $\text{RIGHT}$ then the minimum value of $\text{Rcvd}_r \text{IGHT}$ is sent to the node $\text{left}(\text{RIGHT})$ in at most three rounds.

Proof. We have several cases, depending on the value of variable $\text{state}_r \text{IGHT}$. 

Case 1) $\text{state}_r \text{IGHT} = \text{send}$. We have several sub cases, depending on the value of $\text{state}_l \text{eft}(\text{RIGHT})$.

1.1) If the variable $\text{state}_l \text{eft}(\text{RIGHT}) \in \{\text{left, receive, relay, send}\}$ then Guard $S$ is enabled at node $\text{RIGHT}$ and no other guard is enabled at node $\text{RIGHT}$. Node $\text{RIGHT}$ executes the Guard $S$ and the minimum value of $\text{Rcvd}_r \text{IGHT}$ is sent to its left neighbor $\text{left}_r \text{IGHT}$ in at most one round.

1.ii) If the variable $\text{state}_l \text{eft}(\text{RIGHT}) = \text{right}$ then in finite number of rounds (at most two rounds) Guards $S$ or $SR$ of the $\text{RIGHT}$ node will be enabled (Property 4). One of the enabled guards is arbitrarily selected and will change the variable $\text{state}_l \text{eft}(\text{RIGHT})$ to $\text{receive}$. Then we apply Case 1.i).

Case 2) $\text{state}_r \text{IGHT} = \text{receive}$. Then either Guard $R.Re$ or $RS$ may be enabled at node $\text{RIGHT}$ (but one of them is certainly enabled and no other guard is enabled).

2.i) If Predicate $\text{to}_r \text{c}_v \text{.left}_m \text{sg}(\text{RIGHT})$ is true at node $\text{RIGHT}$, then Guard $R.Re$ is enabled at node $\text{RIGHT}$ and it is the only enabled guard. In at most one round, node $\text{RIGHT}$ executes Guard $R.Re$ and the variable $\text{state}_r \text{IGHT}$ becomes $\text{relay}$. Then Case 3) is applied.
2.ii) If Predicate $to\_rcv\_left\_msg(RIGHT)$ is false at node $RIGHT$, then depending on the variable $state.left(RIGHT)$ Guard $RS$ may be enabled and it is the only enabled guard

2.ii.a) Variable $state.left(RIGHT) \neq send$. Then Guard $RS$ is enabled and it is the only one. In at most one round, node $RIGHT$ executes Guard $RS$ and the variable $state.RIGHT$ becomes $send$. Then Case 1) is applied.

2.ii.b) Variable $state.left(RIGHT) = send$. Then Predicate $enable\_send\_right$ is true at node $left(RIGHT)$. Guard $S$ is selected and executed in at most one round. Variable $state.left(RIGHT)$ becomes $receive$ and Case a) is applied.

Case 3) $state.RIGHT = relay$. Depending upon the $state.left(RIGHT)$ we have several cases:

3.i) If the variable $state.left(RIGHT) = receive$ then Guard $Re.S$ will be enabled at node $RIGHT$ and it is the only enabled guard. It will be executed in at most one round and $state.RIGHT$ becomes $send$. Then Case 1) is applied.

3.ii) If the variable $state.left(RIGHT) = send$ then Guard $Re.R$ will be enabled at node $RIGHT$ and it is the only enabled guard. It will be executed in at most one round and $state.RIGHT$ becomes $receive$. Then Case 2) is applied. Since the number of elements in every node is finite and the number of nodes in the network is finite, then after a while Predicate $to\_rcv\_left\_msg(RIGHT)$ is not true anymore. Thus Guard $R.Re$ will not be enabled when the variable $state.RIGHT = receive$. The only enabled Guard will be $RS$, then Case 1) will apply.

3.iii) If the variable $state.left(RIGHT) = relay$ then Guard $ReS$ is enabled at node $RIGHT$ and it is the only enabled guard. It will be executed in at most one round and $state.RIGHT$ becomes $send$. Then Case 1) is applied.
3.iv) If the variable \( \text{state.left(RIGHT)} = \text{left} \) then at node \( \text{left(RIGHT)} \) one of the Guards \( L.R, L.S, \) or \( L.Re \) is enabled at node \( \text{left(RIGHT)} \). One of the enabled guards is selected and executed at node \( \text{left(RIGHT)} \) in at most one round. Variable \( \text{state.left(RIGHT)} \) becomes either \textit{relay} (and then Case 3) is applied), \textit{send} (and then Case 1) is applied), or \textit{receive} (and then Case 2) is applied). If the variable \( \text{state.RIGHT} \) is \textit{relay}, then either Case 3.ii) or Case 3.iii) is applied in at most one more round.

3.v) If the variable \( \text{state.left(RIGHT)} = \text{right} \) then one of the Guards \( Ri.Re, R.S, \) or \( R.R \) is enabled at node \( \text{left(RIGHT)} \). One of the enabled guards is selected and executed in at most one round. In at most one round the variable \( \text{state.left(RIGHT)} \) becomes either \textit{relay} (and then Case 3) is applied), \textit{send} (and then Case 1) is applied), or \textit{receive} (and then Case 2) is applied). Then if the variable \( \text{state.RIGHT} \) is \textit{relay} then then either Case 3.ii) or Case 3.iii) is applied in at most one more round.

Case 4) \( \text{state.RIGHT} = \text{left} \). Then Guard \( LS \) is the only enabled guard at node \( RIGHT \). It will be selected and executed in at most one round, and variable \( \text{state.RIGHT} \) becomes \textit{send}. Then Case 1) is applied. Case 5) \( \text{state.RIGHT} = \text{right} \). Then Guard \( Ri.S \) is the only enabled guard at the node \( RIGHT \). It will be selected and executed in at most one round, and the variable \( \text{state.RIGHT} \) becomes \textit{send}. Then Case 1) is applied. □

Property 5. If some value in the set \( \text{Rcvd.RIGHT} \) is lesser than some value in the set \( \text{Rcvd.left(RIGHT)} \) (i.e., \( \min(\text{Rcvd.RIGHT}) < \max(\text{Rcvd.left(RIGHT)}) \)) then the value will be eventually moved from \( \text{Rcvd.RIGHT} \) to \( \text{Rcvd.left(RIGHT)} \).

Proof. Let \( x \) be the minimum in \( \text{Rcvd.RIGHT} \) (i.e. \( x \leq \{ \text{Rcvd.RIGHT} \setminus x \} \)) and \( y \) be the maximum in \( \text{Rcvd.right(LEFT)} \) (i.e. \( y \geq \{ \text{Rcvd.left(RIGHT)} \setminus y \} \)). We have that \( x < y \). Predicate \textit{to_send_min(RIGHT)} is true at node \( RIGHT \) and
Predicate 

to\textunderscore send\_max(left(RIGHT)) is true at node left(RIGHT). In Property 4 we show 
that if Predicate to\textunderscore send\_min(RIGHT) is true at node RIGHT then value x is re-
moved from Rcvd.RIGHT (Macro send\_Min(RIGHT) is executed at node RIGHT) 
in finite number of rounds (at most three rounds). The value which may be sent by 
node left(RIGHT) is y and since 
y \leq \{Rcvd.left(RIGHT) \setminus y\} and x < y, it implies that by adding y to 
\{Rcvd.(RIGHT) \setminus x\} we converge to the situation when \( \min(Rcvd.RIGHT) \geq \max(Rcvd.left(RIGHT)) \). □

Property 6. If for the node i = RIGHT the condition: “any value in Rcvd.i is 
greater or equal to any value in Rcvd.left(i) i.e., \( \min(Rcvd.i) \geq \max(Rcvd.left(i)) \)“, 
then no new messages are generated by node RIGHT.

Proof. If the condition is true, then Predicate to\textunderscore send\_min(RIGHT) is false, thus 
Guard S is disabled at node RIGHT. Guard S is the only guard whose execution 
can generate new messages from node RIGHT. □

Property 7. If Predicates to\textunderscore send\_max(MIDDLE) and to\textunderscore send\_min(MIDDLE) 
are true, and the variable state.MIDDLE = send, then in at most one round, the 
minimum and the maximum values from Rcvd.MIDDLE are sent accordingly.

Proof. Since Predicates to\textunderscore send\_max and to\textunderscore send\_min are true, Guard S may be 
true if the variable state.left(MIDDLE) ∈ \{receive, left, relay, send\} 
or if the variable state.right(MIDDLE) ∈ \{receive, right, relay, send\}. We have 
then the following cases:

1) The variables state.left(MIDDLE) ∈ \{receive, left, relay, send\} and 
state.right(MIDDLE) ∈ \{receive, right, relay, send\}. Then in at most one 
round the node MIDDLE will execute Guard S, the minimum and the maxi-
mum values from Rcvd.MIDDLE are sent accordingly, and change the variable 
state.MIDDLE to receive.

35

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
2) The variable $state.left(MIDDLE) = right$, then the node $left(MIDDLE)$ cannot be node RIGHT. The node $left(MIDDLE)$ can be either node LEFT or other MIDDLE node. If the node $left(MIDDLE)$ is the node LEFT, then Guard $Ri.S$ is the only enabled guard at node $left(MIDDLE)$. After the execution of Guard $Ri.S$ the variable $state.left(MIDDLE)$ is changed to $send$. We then apply Case 1).

If the node $left(MIDDLE)$ is not node LEFT, then node $left(MIDDLE)$ cannot be node RIGHT and obviously node $left(MIDDLE)$ is another MIDDLE node. Then node $left(MIDDLE)$ can execute either Guards $Ri.Re$, $R.S$, $R.R$ depending upon the predicates $to.rcv.right.msg(MIDDLE)$ and $to.rcv.left.msg(MIDDLE)$. The variable $state.left(MIDDLE)$ will then change in at most one round to either $relay$, $receive$ or $send$. We then apply Case 1).

3) The variable $state.right(MIDDLE) = left$, then the node $right(MIDDLE)$ cannot be node LEFT. The node $right(MIDDLE)$ can be either a MIDDLE node or node RIGHT. If node $right(MIDDLE)$ is node RIGHT, then Guard $LS$ is the only enabled guard. It will be executed in at most one round and the variable $state.right(MIDDLE)$ will be changed to $send$. Then Case 1) applies. If right(MIDDLE) node is some other MIDDLE node, depending upon the predicates $to.rcv.right.msg(MIDDLE)$ and $to.rcv.left.msg(MIDDLE)$ the variable $state.right(MIDDLE)$ is changed from $left$ to $relay$. Then Case 1) applies.

□

Property 8. If Predicates $to.send.max(MIDDLE)$ and $to.send.min(MIDDLE)$ are true, and the variable $state.MIDDLE = receive$, then in finite rounds, the variable $state.MIDDLE$ becomes $send$. 

36

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Proof. We have the following cases:

1) If Predicates \( \text{to.rcv.right.msg}(\text{MIDDLE}) \) and \( \text{to.rcv.left.msg}(\text{MIDDLE}) \) are true, then Guard \( R.Re \) will be enabled. In at most one round then the variable state.MIDDLE is changed to relay. Then at least of the guards \( Re.S1, Re.S2, Re.R1 \) or \( Re.R2 \) is enabled. Depending upon the variables state.left(MIDDLE) and state.right(MIDDLE), the variable state.MIDDLE becomes either receive or send in at most one round. If the variable state of the MIDDLE node is changed to receive, then in finite amount of rounds it will be changed to send.

2) If Predicate \( \text{to.rcv.right.msg}(\text{MIDDLE}) \) is true and \( \text{to.rcv.left.msg}(\text{MIDDLE}) \) is false then in at most one round the variable state.MIDDLE changes to right. Since Predicate \( \text{to.rcv.left.msg}(\text{MIDDLE}) \) is false and \( \text{to.rcv.right.msg}(\text{MIDDLE}) \) is true then Guards \( L.S \) and \( L.R \) will be enabled. One of the enabled guards will be selected arbitrarily.

   If Guard \( L.S \) is selected then the variable state.MIDDLE is changed to send.

   If Guard \( L.R \) is selected then the variable state.MIDDLE is changed to receive. Then Guard \( R.Ri \) is the only enabled guard and it changes the variable state.MIDDLE to right in at most one round. Then either Guards \( R.S \) or \( R.R \) will be enabled. Arbitrarily one of the enabled guard is selected. If Guard \( R.S \) is selected, then the variable state.MIDDLE is changed to send in at most three rounds. If Guard \( R.R \) is selected, then the variable state.MIDDLE is changed to receive. Then in finite number of rounds variable state.MIDDLE will be changed to send.

3) If Predicate \( \text{to.rcv.right.msg}(\text{MIDDLE}) \) is false and \( \text{to.rcv.left.msg}(\text{MIDDLE}) \) is true then the variable state.MIDDLE is changed to left. Then node MIDDLE may execute either Guard \( L.S \) or \( L.R \). Arbitrarily one of the guards
is selected. If Guard $L.S$ is selected then in at most two rounds the variable $state.MIDDLE$ is changed to $send$. If Guard $L.R$ is selected then in at most one round the variable $state$ changes to $receive$. Then in finite number of rounds variable $state.MIDDLE$ will be changed to $send$.

\[\square\]

**Property 9.** If Predicates $tosend\_max(MIDDLE)$ and $tosend\_min(MIDDLE)$ are true, then in finite number of rounds the minimum and the maximum values from $Rcvd.MIDDLE$ are sent accordingly.

**Proof.** We have several cases, depending on the value of the variable $state.MIDDLE$.

a.i) $state.MIDDLE = send$. By Property 7, in at most one round the minimum and the maximum values from $Rcvd.MIDDLE$ are sent accordingly.

a.ii) $state.MIDDLE = receive$. By Property 8, in finite number of rounds the variable $state$ of the MIDDLE node is changed to $send$. Then Case a.i) applies.

a.iii) $state.MIDDLE = relay$. Depending upon variable $state$ of the left node and the right node one of the Guards $Re.S1$, $Re.S2$, $Re.R1$ or $Re.R2$ will be enabled and after the execution of the enabled guard the variable $state$ of the MIDDLE node is changed to $send$ or $receive$ in at most one round. If the $state.MIDDLE$ changes to $send$ then apply Case a i). If $state.MIDDLE$ becomes $receive$ then the middle applies Case a ii).

a.iv) $state.MIDDLE = left$. If the predicate $to\_rcv\_right\_msg(MIDDLE)$ is true then Guard $L.Re$ is the only enabled guard and changes the variable $state(MIDDLE)$ to $relay$. This then applies is case iii). If that predicate is false then Guard $L.S$ is the only guard enabled and it changes the variable $state.MIDDLE$ to $send$. If the predicate $to\_rcv\_left\_msg(MIDDLE)$ to $send$. If the predicate $to\_rcv\_left\_msg(MIDDLE)$ is true then Guard $L.R$ is the
only enabled guard and it changes the variable state of the MIDDLE node to receive in at most one round. This then follows case ii).

a.v) state.MIDDLE = right. If the predicate to.rcv.left.msg(MIDDLE) is true then Ri.Re is the only enabled guard and changes the variable state(MIDDLE) to relay. This then applies case iii). If that predicate is false then Rs is the only guard enabled and it changes the variable state.MIDDLE to send. If the predicate to.rcv.left.msg(MIDDLE) to send. If the predicate to.rcv.right.msg(MIDDLE) is true then R.R is the only enabled guard and it changes the variable state of the MIDDLE node to receive in at most one round. This then follows case ii) and in finite number of rounds the variable state of the MIDDLE node will be changed to send which will eventually send its maximum and/or minimum element.

□

Property 10. If Predicate to.send.max is true, Predicate to.send.min is false, and the variable state.MIDDLE = send, then in at most two rounds the maximum value of Rcvd.MIDDLE is sent to the node right(MIDDLE).

Proof. Depending upon the variable state.right(MIDDLE), Predicate enable.send.right at node MIDDLE may be true or false.

1) If the variable state.right(MIDDLE) ∈ {receive, right, relay, send}, then Predicate enable.send.right at node MIDDLE is true, then Guard SR is enabled at node MIDDLE. The maximum element in the set Rcvd.MIDDLE will be sent to the the node right(MIDDLE). In at most one round the variable state.MIDDLE will be changed to receive.

2) If the state.right(MIDDLE) = left then Predicate enable.send.right at node MIDDLE is false. The node right(MIDDLE) cannot be a LEFT node. It can be either a MIDDLE node or node RIGHT.

39
If node right(MIDDLE) is node RIGHT, since the variable state.right(MIDDLE) = left, then Guard LS is the only enabled guard at node right(MIDDLE). In at most one round the variable state.right(MIDDLE) becomes send. Then Case 1) of the RIGHT node follows.

If node right(MIDDLE) is some other MIDDLE node, depending upon Predicates to.recv.right.msg(MIDDLE) and to.recv.left.msg(MIDDLE), the variable state.right(MIDDLE) is changed from left to relay in at most one round. Then Case 1) applies.

Property 11. If Predicate to.send.max is true, Predicate to.send.min is false, and the variable state.MIDDLE = receive, then in finite number of rounds the variable state.MIDDLE will be changed to send.

Proof. Depending upon Predicates to.recv.right.msg(MIDDLE) and to.recv.left.msg(MIDDLE), the variable state.MIDDLE will be changed to relay, right or left.

1) If Predicate to.recv.right.msg(MIDDLE) and to.recv.left.msg(MIDDLE) are true, then Guard R.Re will be enabled. In at most one round, the variable state.MIDDLE is changed to relay. Then one of the Guards Re.S1, Re.S2, Re.R1, or Re.R2 will be enabled. Depending upon the variables state.left(MIDDLE) and state.right(MIDDLE), the variable state of the MIDDLE node is either changed to receive or send in at most one round. If the variable state.MIDDLE is changed to receive, then in finite number of rounds the variable state.MIDDLE will be changed to send.

2) If Predicates to.recv.right.msg(MIDDLE) is true and to.recv.left.msg(MIDDLE) is false, then Guards L.S and L.R will be enabled. One of the enabled guards will be selected arbitrarily.
If Guard $L.S$ is selected then variable $state.MIDDLE$ is changed to $send$.

If Guard $L.R$ is selected then variable $state.MIDDLE$ is changed to $receive$. Then Guard $R.Ri$ is the only enabled guard. It will be executed in at most one round, and the variable $state.MIDDLE$ changes to $right$. Then either Guard $R.S$ or $R.R$ will be enabled. Arbitrarily one of the enabled guard is selected.

If Guard $R.S$ is selected, then the variable $state.MIDDLE$ is changed to $send$ in at most three rounds.

If Guard $R.R$ is selected then the variable $state.MIDDLE$ is changed to $receive$. Then in finite number of rounds variable $state.MIDDLE$ will be changed to $send$.

3) If Predicates $to.rcv.right.msg(MIDDLE)$ is false and $to.rcv.left.msg(MIDDLE)$ is true. Then the node $MIDDLE$ may execute either Guard $L.S$ or $L.R$. Arbitrarily one of the guards is selected.

If Guard $L.S$ is selected then in at most two rounds the variable $state.MIDDLE$ is changed to $send$.

If Guard $L.R$ is selected, the variable $state$ changes to $receive$. Then in finite number of rounds the variable $state$ of the $MIDDLE$ node is changed to $send$.

□

Property 12. If Predicate $tosend.max$ is true and Predicate $tosend.min$ is false, then in finite number of rounds the maximum value in the set $Rcvd.MIDDLE$ is sent to the node right($MIDDLE$).

Proof. We have several cases:

b.i) $state.MIDDLE = send$. By Property 10, in at most two rounds the maximum value of $Rcvd.MIDDLE$ is sent to the node right($MIDDLE$).
b.ii) \( \text{state.MIDDLE} = \text{receive} \). By Property 11, in finite number of rounds the variable \( \text{state.MIDDLE} \) will be changed to \text{send}. Then Case b.i) applies.

b.iii) \( \text{state.MIDDLE} = \text{relay} \). Depending upon the variables \( \text{state.left(MIDDLE)} \) and \( \text{state.right(MIDDLE)} \) one of the Guards \( \text{Re.S1}, \text{Re.S2}, \text{Re.R1} \) or \( \text{Re.R2} \) will be enabled. After the execution of the enabled guard, the variable \( \text{state.MIDDLE} \) is changed to \text{send} or \text{receive} in at most one round. If the variable \( \text{state.MIDDLE} \) changes to \text{send} then Case a.i) of Property 9 applies. If the variable \( \text{state.MIDDLE} \) becomes \text{receive} then Case a.ii) of Property 9 applies.

b.iv) \( \text{state.MIDDLE} = \text{left} \).

If Predicate \( \text{to.recv.right.msg(MIDDLE)} \) is \text{true}, then Guard \( \text{L.Re} \) is the only enabled guard at node MIDDLE. It is executed in at most one round, and the variable \( \text{state.MIDDLE} \) becomes \text{relay}. We then apply Case b.iii).

If Predicate \( \text{to.recv.right.msg(MIDDLE)} \) is \text{false} then Guard \( \text{L.S} \) is the only guard enabled at node MIDDLE. It is executed in at most one round, and the variable \( \text{state.MIDDLE} \) changes to \text{send}.

If Predicate \( \text{to.recv.left.msg(MIDDLE)} \) is \text{true}, then Guard \( \text{L.R} \) is the only enabled guard at node MIDDLE. It is executed in at most one round, and the variable \( \text{state.MIDDLE} \) is changed to \text{receive}. Then Case b.ii) follows.

b.v) \( \text{state.MIDDLE} = \text{right} \).

If Predicate \( \text{to.recv.left.msg(MIDDLE)} \) is \text{true}, then Guard \( \text{Ri.Re} \) is the only enabled guard at node MIDDLE. It is executed in at most one round, and the variable \( \text{state.MIDDLE} \) is changed to \text{relay}. Then Case b.iii) applies.

If Predicate \( \text{to.recv.left.msg(MIDDLE)} \) is \text{false}, then Guard \( \text{R.S} \) is the only enabled guard at node MIDDLE. It is executed in at most one round, and the variable \( \text{state.MIDDLE} \) is changed to \text{send}.
If Predicate $to_{rcv\_right\_msg}(MIDDLE)$ is true then Guard $R.R$ is the only enabled guard at node MIDDLE. It is executed in at most one round, and the variable $state.MIDDLE$ is changed to receive. Then Case b.ii) applies.

□

Property 13. If Predicate $to_{send\_max}$ is false, Predicate $to_{send\_min}$ is true, and the variable $state.MIDDLE = send$, then in finite number of rounds the minimum value in the set $Rcvd.MIDDLE$ is sent to the node $left(MIDDLE)$.

Proof. We prove the above property by taking the following cases:

1) If the variable $state.left(MIDDLE) \in \{receive, left, relay, send\}$ then Predicate $enable_{send.left}$ is true at node MIDDLE, and Guard $S$ is the only enabled guard at node MIDDLE. It will be executed in at most one round, the minimum value in the set $Rcvd.MIDDLE$ is sent to the node $left(MIDDLE)$, and the variable $state.MIDDLE$ changes to receive.

2) If the variable $state.left(MIDDLE) = right$, then Predicate $enable_{send.left}$ at node MIDDLE is false. The node $left(MIDDLE)$ cannot be a RIGHT node. It can be either a MIDDLE node or node LEFT.

If node $left(MIDDLE)$ is node LEFT, since the variable $state.left(MIDDLE) = right$, then Guard $Ri.S$ is the only enabled guard at node $left.MIDDLE$. In at most one round the variable $state.left(MIDDLE)$ becomes send. Then Case 1) of the LEFT node follows.

If node $left(MIDDLE)$ is some other MIDDLE node, depending upon Predicates $to_{rcv\_right\_msg}(MIDDLE)$ and $to_{rcv\_left\_msg}(MIDDLE)$, the variable $state.left(MIDDLE)$ is changed from right to relay in at most one round. Then Case 1) applies.

□
Property 14. If Predicate to_send_max is false, Predicate to_send_min is true, and
the variable state.MIDDLE = receive, then in finite number of rounds the variable
state.MIDDLE becomes send.

Proof. We have the following cases:

1) If Predicates to_rv.right_msg(MIDDLE) and to_rv.left_msg(MIDDLE)
    are true, then Guard R.Re is the only enabled guard. It will be executed
    in at most one round, and the variable state.MIDDLE is changed to relay.
    One of the Guards Re.S1, Re.S2, Re.R1 or Re.R2 is enabled, depending upon
    the variables state.left(MIDDLE) and state.right(MIDDLE). The variable
    state.MIDDLE becomes either receive or send in at most one round.
    If the variable state.MIDDLE becomes receive, then in finite number of
    rounds it will be changed to send.

2) If Predicates to_rv.right_msg(MIDDLE) is true and to_rv.left_msg(MIDDLE)
    is false, then Guards L.S and/or L.R may be enabled. One of the enabled
    guards will be selected arbitrarily.
    If Guard L.S is selected then the variable state.MIDDLE is changed to send.
    If Guard L.R is selected then the variable state.MIDDLE is changed to
    receive. Then Guard R.Ri is the only enabled guard at node MIDDLE. In at
    most one round is executed and the variable state.MIDDLE becomes right.
    Then either Guards R.S or R.R will be enabled. Arbitrarily one of the enabled
    guards is selected.
    If Guard R.S is selected then variable state.MIDDLE is changed to send in
    at most three rounds.
    If Guard R.R is selected then the variable state.MIDDLE is changed to
    receive. Then in finite number of rounds variable state will be changed to
    send.
3) If Predicate $to_{rcv.right}.msg(MIDDLE)$ is false and $to_{rcv.left}.msg(MIDDLE)$ is true, then either Guard $LS$ or $LR$ is enabled. Arbitrarily one of the guards will be selected. If Guard $LS$ is selected then it is executed in at most one round. Then in at most two rounds the variable $state.MIDDLE$ is changed to $send$.

If Guard $LR$ is selected, then it is executed in at most one round. The variable $state.MIDDLE$ is changed to $receive$. Then in finite number of rounds the variable $state.MIDDLE$ is changed to $send$.

\[\square\]

**Property 15.** If Predicates $to_{send}.max$ is false and Predicate $to_{send}.min$ is true, then in finite number of rounds the minimum value in the set $Rcvd.MIDDLE$ is sent to the node $left(MIDDLE)$.

**Proof.** We have several cases:

1. **c.i)** $state.MIDDLE = send$. By Property 13, in finite number of rounds the minimum value in the set $Rcvd.MIDDLE$ is sent to the node $left(MIDDLE)$.

2. **c.ii)** $state.MIDDLE = receive$. By Property 14, in finite number of rounds the variable $state.MIDDLE$ becomes $send$. Then Case c.i) applies.

3. **c.iii)** $state.MIDDLE = relay$. Depending upon the variable $state.left(MIDDLE)$ and $state.right(MIDDLE)$ one of Guards $Re.S1$, $Re.S2$, $Re.R1$ or $Re.R2$ will be enabled. After the execution of the selected guard, the variable $state.MIDDLE$ is changed to either $send$ or $receive$ in at most one round. If the state.MIDDLE changes to $send$ then Case a.i) of Property 9 applies. If state.MIDDLE becomes $receive$ then Case a.ii) of Property 9 applies.

4. **c.iv)** $state.MIDDLE = left$.
If Predicate \( \text{to\.right\.msg}(MIDDLE) \) is \textit{true} then Guard \( L\, Re \) is the only enabled guard. It is selected and executed in at most one round. The variable \( state(MIDDLE) \) changes to \textit{relay}. Then Case c.iii) applies.

If Predicate \( \text{to\.right\.msg}(MIDDLE) \) is \textit{false} then Guard \( L\, S \) is the only guard enabled. It is selected and executed in at most one round. The variable \( state(MIDDLE) \) is changed to \textit{send}. Then Case c.i) applies.

If Predicate \( \text{to\.left\.msg}(MIDDLE) \) is \textit{true} then Guard \( L\, R \) is the only enabled guard. It is selected and executed in at most one round. The variable \( state(MIDDLE) \) is changed to \textit{receive} in at most one round. Then Case c.ii) applies.

\textbf{c.v) state(MIDDLE) = right.}

If Predicate \( \text{to\.left\.msg}(MIDDLE) \) is \textit{true} then Guard \( R\, R \) is the only enabled guard. It is selected and executed in at most one round. The variable \( state(MIDDLE) \) is changed to \textit{receive} in at most one round. Then Case c.ii) applies.

Property 16. If some value in \( Rcvd.i \) is lesser than some value in \( Rcvd.left(RIGHT) \) (i.e., \( \min(Rcvd.i) < \max(Rcvd.left(i)) \)) then the value will be eventually moved from \( Rcvd.i \) to \( Rcvd.left(i) \).

If some value in \( Rcvd.i \) is larger than some value in \( Rcvd.right(i) \) (i.e., \( \max(Rcvd.i) > \min(Rcvd.right(i)) \)) then the value will be eventually moved from
Proof. We have several cases:

a) Predicates \( to\_send\_max(MIDDLE) \) and \( to\_send\_min(MIDDLE) \) are true. By Property 9, in finite number of rounds the minimum and the maximum values from the set \( Rcvd.MIDDLE \) are sent accordingly.

b) Predicate \( to\_send\_max \) is true and Predicate \( to\_send\_min \) is false. By Property 12, in finite number of rounds the maximum value in the set \( Rcvd.MIDDLE \) is sent to the node \( right(MIDDLE) \).

c) Predicate \( to\_send\_max \) is false and Predicate \( to\_send\_min \) is true. By Property 15, in finite number of rounds the minimum value in the set \( Rcvd.MIDDLE \) is sent to the node \( left(MIDDLE) \).

Property 17. If for some \( MIDDLE \) node \( i \), \(|Rcvd.i| > 1\), in finite number of steps variable \( Rcvd.i \) has only one value \(|Rcvd.i| = 1\).

Proof. Predicate \( to\_send\_max(i) \) is true. Using the proof of Property 16, in finite number of rounds, the maximum value in \( Rcvd.i \) is sent to node \( right(i) \).

Property 18. If for the node \( i = MIDDLE \) the condition: "\(|Rcvd.i| = 1 \) and any value in \( Rcvd.i \) is less or equal to than any value in \( Rcvd.right(i) \) and greater or equal to any value in \( Rcvd.left(i) \): \( \max(Rcvd.i) \leq \min(Rcvd.right(i)) \land \min(Rcvd.i) \geq \max(Rcvd.left(i)) \)", then no new messages are generated by node \( i \).

Proof. If the condition is true, then both Predicates \( to\_send\_min(i) \) and \( to\_send\_max(i) \) are false. Thus Guard \( S \) is disabled at node \( MIDDLE \). Guard \( S \) is the only guard whose execution can generate new messages from node \( RIGHT \).
Lemma 5. (Convergence) Starting from some arbitrary configuration $C_0$, in finite number of rounds the algorithm reaches a configuration $C$ that satisfies Predicate $\mathcal{P}$.

Proof. Depending on node $i$, we have three cases:

a) For the \textit{LEFT} node. Let $i = \text{LEFT}$.

In Property 2 we show that if $|\text{Rcvd}.i| > 1$, in finite number of steps variable $\text{Rcvd}.i$ has only one value $|\text{Rcvd}.i| = 1$.

In Property 3 we show that after finite number of rounds, any value in $\text{Rcvd}.i$ is less or equal to than any value in $\text{Rcvd}.right(i)$: $\max(\text{Rcvd}.i) \leq \min(\text{Rcvd}.right(i))$.

In Property 3 we show that if node $i$ satisfies the condition:

$|\text{Rcvd}.i| = 1$ and any value in $\text{Rcvd}.i$ is less or equal to than any value in $\text{Rcvd}.right(i)$: $|\text{Rcvd}.i = 1|weedemin(\text{Rcvd}.i) \leq \min(\text{Rcvd}.right(i))$

then no new messages are generated by node $i$.

b) For a \textit{MIDDLE} node $i$.

In Property 17 we show that if $|\text{Rcvd}.i| > 1$, in finite number of steps variable $\text{Rcvd}.i$ has only one value $|\text{Rcvd}.i| = 1$.

In Property 16 we show that after finite number of rounds, any value in $\text{Rcvd}.i$ is less or equal to than any value in $\text{Rcvd}.right(i)$ and greater or equal to any value in $\text{Rcvd}.left(i)$: $\max(\text{Rcvd}.i) \leq \min(\text{Rcvd}.right(i)) \land \min(\text{Rcvd}.i) \geq \max(\text{Rcvd}.left(i))$.

In Property 18 we show that if node $i$ satisfies the condition:

$|\text{Rcvd}.i| = 1$ and any value in $\text{Rcvd}.i$ is less or equal to than any value in $\text{Rcvd}.right(i)$ and greater or equal to any value in $\text{Rcvd}.left(i)$: $\max(\text{Rcvd}.i) \leq \min(\text{Rcvd}.right(i)) \land \min(\text{Rcvd}.i) \geq \max(\text{Rcvd}.left(i))$

then no new messages are generated by node $i$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
c) For the RIGHT node. Let $i = \text{RIGHT}$.

In Property 5 we show that any value in $Rcvd.i$ is greater or equal to any value in $Rcvd.left(i)$, $\min(Rcvd.i) \geq \max(Rcvd.left(i))$.

In Property 6 we show that if node $i$ satisfies the condition: any value in $Rcvd.i$ is greater or equal to any value in $Rcvd.left(i)$, $\min(Rcvd.i) > \max(Rcvd.left(i))$

Then no new messages are generated by node $i$.

4.3 Proof of our algorithm if the system starts from a correct state

We prove that our algorithm works fine by taking an example. Consider the Figure 2 which has three nodes $P_1$ or the LEFT node, $P_3$ or the RIGHT node, $P_2$ the MIDDLE node. Assume that

- There are no faults occurring in the system.
- The LEFT node has the $Rcvd$ set with the values 6, 5 and with the $state$ variable equal to $send$.
- The MIDDLE node has the $Rcvd$ set with the value 0 and also with the $state$ variable equal to $receive$.
- The RIGHT node has the value of $state$ variable equal to $receive$ and the $Rcvd$ set equal to empty set.

After one round: The Guard $S$ of the node LEFT enabled and $send\_Max(i)$ macro is executed and the value 6 is sent to its right neighbor. After sending the maximum value to its right neighbor the variable $state$ of the node LEFT node is changed to $receive$. None of the guards are enabled at the nodes RIGHT and MIDDLE.

After another round: At the node MIDDLE node Guard $R\_L$ is enabled and it receives the element from its left neighbor and changes its variable $state$ to $left$. None of the guards are enabled at the nodes LEFT and the RIGHT.

After one more round: The leftmost node (the node LEFT) and the rightmost node (node RIGHT) will not have any guards enabled. At the node MIDDLE the
Guard $L_S$ will be enabled and the state variable of the node MIDDLE is changed to send.

After one more round: At the node MIDDLE Guard $S$ will be enabled and it will send its maximum element to its right neighbor and the minimum element to
its left neighbor also changes its state variable to receive. The nodes LEFT and the RIGHT will not have any enabled guards.

After one more round: R_Re will be the only enabled guard at the leftmost(LEFT) and the rightmost(RIGHT) node. The node LEFT receives the element from its right neighbor and changes its variable state to relay. In the same way the node RIGHT receives the element from its left neighbor and changes its variable state to relay. The node MIDDLE will not have any enabled guards, so it stays in the previous state.

After one more round: The Guard Re_S of the nodes LEFT and RIGHT will be enabled and this will change the variable state of the nodes LEFT and RIGHT to send. The MIDDLE node will not have any enabled guards and so will remain in the previous state.

After one more round: The Guard S of the LEFT node will be enabled and it will send its maximum element to its right neighbor and will also change its state variable to receive. The Guard SR of the rightmost node will be enabled and will change the state variable to receive. The node MIDDLE remains in the previous state, since there are no enabled guards.

After one more round: The Guard R_L of the node MIDDLE will be enabled and the node MIDDLE will receive elements from its left neighbor and also will change the variable state to left. The nodes LEFT and the RIGHT will not have any enabled guards and so will remain in the previous state.

After one more round: The leftmost and the rightmost nodes will not have any enabled guards. The node MIDDLE will have the Guard L_S enabled and will change its variable state to send.

After this round all the nodes will just keep changing its variable state (until a fault occurs and changes the state of the system) without generating any new messages.
4.4 Proof of our algorithm if the system starts from an incorrect state

Consider the Figure 3. It has three nodes leftmost node(LEFT), rightmost node(RIGHT), and a MIDDLE node with the following specifications. The LEFT node’s state variable is receive and its Rcvd set variable is 5,6. The RIGHT node’s Rcvd set variable is an empty set and its state variable is receive. The MIDDLE nodes state variable is receive and its Rcvd variable is 0.

After one round: The Guard RS of the leftmost node will be enabled and will change the variable state of the LEFT node to send. The RIGHT and the MIDDLE nodes remain in their previous state. This then follows the same steps as if the system had started from the correct states (provided it is not disturbed by any faults) and in finite number of steps the values will be in sorted order. If there are any faults in the middle of the execution the nodes will be applying different set of guards and in finite number of rounds the values will be in the sorted order.

4.4 Space Complexity our Algorithm

Space complexity of a node or a process is the maximum number of bits required to store the variables in that node or a process. Assume that we start with at most $k$
to store the variables in that node or a process. Assume that we start with at most $k$
items in the network Variable

- $\text{Rcvd}.i$ uses $k \times L$ bits where $L$ is the maximum number of bits to store one
item.
- $\text{state}.i$ uses three bits since it make take either one of the five values
left, right, relay, send, receive.
- $M_{\text{tr}}, M_{\text{ls}}, M_{\text{rs}}, M_{\text{rr}}$ uses $L$ bits each.

And so, the number of bits required is $O(k \times L)$ per node.

4.5 Message Complexity our Algorithm

Every node may send at most two messages in a round, each message containing
one item. Hence there will be a total of $2 + 2 \times (n - 2)$ messages. And each message
has at most $L$ bits. Therefore the message complexity per round is $L \times (2n - 2)$. The
total message complexity of network is $O(L \times (2n - 2))$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 5

CONCLUSION AND FUTURE WORK

We have presented a self-stabilizing sorting algorithm using composite atomicity model. The algorithm has three special nodes LEFT, RIGHT, and MIDDLE. The left node sends the maximum element (if exists) to its only neighbor. In the same way the RIGHT node sends the minimum element to its only left neighbor and the MIDDLE node sends both the maximum and the minimum elements to its right and left neighbors respectively. The proposed algorithm guarantees that starting in any arbitrary configuration and in finite number of steps; the values in the nodes are in sorted order. The algorithm is self-stabilizing: it copes with wrong initialization and it adapts to any loss of data that occurs due to memory corruption of the nodes.

Furthermore our algorithm is not time and communication optimal, so there is a room for developing a self-stabilizing sorting with optimal time and message complexity in future.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Lakshmi Visvanathan

Home Address:
4224 Cottage Cirlee Apt 4
Las Vegas, NV 89119

Degree:
Bachelor of Science,
Bharadhidasan University

Special Honors and Awards:
Graduate Assistantship/Teaching Assistantship
(School of Computer Science)

Thesis Title: Self-Stabilizing Sorting on Linear Network
Thesis Examination Committee:
Chairperson, Dr. Ajoy K. Datta, Ph. D.
Committee Member, Dr. Wolfgang Bein, Ph. D.
Committee Member, Dr. Yoo-hwan Kim, Ph. D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph. D.