Self-stabilizing distributed algorithms for acyclic graphs

Viruthagiri Natarajan
University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/386

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Self-stabilizing distributed algorithms for acyclic graphs

Natarajan, Viruthagiri, M.S.
University of Nevada, Las Vegas, 1994
SELF-STABILIZING DISTRIBUTED
ALGORITHMS FOR ACYCLIC GRAPHS

by

Viruthagiri Natarajan

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
August 1994
The thesis of Viruthagiri Natarajan for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kumar Datta, Ph.D

Examining Committee Member, Kazem Taghva, Ph.D

Examining Committee Member, Laxmi Gewali, Ph.D

Graduate Faculty Representative, Ashok Iyer, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas
August 1994
ABSTRACT

A self-stabilizing distributed system is a network of processors, which when started from an arbitrary and possibly illegal state, always returns to a legal state in a finite number of steps. Two self-stabilizing protocols for distributed systems are presented in this thesis. The first protocol topologically sorts the processors in a distributed system of directed acyclic graph (DAG) topology and uses this information to build a shortest path routing table in each node in the system to all accessible nodes from that node. The second protocol determines the rank of the individual processors in a distributed tree network based on the values possessed by them. Due to the self-stabilizing nature of these protocols the system can withstand transient errors and recover automatically from them.
Contents

Abstract ................................................................. iii
Acknowledgments ..................................................... vi

1 INTRODUCTION ......................................................... 1
1.1 Notations ............................................................ 3

2 SHORTEST PATH ALGORITHM ................................................. 4
2.1 The Algorithms .................................................... 5
  2.1.1 The Topological Sort Algorithm ......................... 5
  2.1.2 The Shortest Path Algorithm ............................. 7
2.2 Proofs of Correctness ................................................ 9
  2.2.1 Topological Sort ............................................. 9
  2.2.2 The Shortest Path Algorithm ............................. 10
2.3 Summary ............................................................. 12

3 RANKING ALGORITHM ....................................................... 13
3.1 The Ranking Algorithm ........................................... 13
  3.1.1 Description of the protocol ............................... 13
  3.1.2 Pre-processing stage ....................................... 17
  3.1.3 Ranking stage .............................................. 18
  3.1.4 Error correction ........................................... 21
3.2 Proof of correctness ................................................. 22
3.3 Summary ............................................................. 24
ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Ajoy Kumar Datta for helping and encouraging me throughout the course of this thesis and my degree program. I am indebted to him for the support he provided. I would also like to thank the Computer Science department for having supported me as a teaching assistant. Special thanks to Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Ashok Iyer for serving on my committee.

This thesis is dedicated to my father who dreamed of me having a graduate degree, but who is not alive today to see his dream come true and to my mother, both of whom supported and encouraged me throughout my life. Also, I would like to thank my sisters for their care and love. And last, but certainly not least, I would like to thank my wife, Vijju, for being there for me while I was completing my research.
Chapter 1

INTRODUCTION

A distributed system consists of a set of loosely connected machines. These machines communicate with each other through shared memory and/or message passing in order to achieve a common goal. The global state of the system is the cross product of all the local states of the machines in the network. Depending on the way the machines are connected in the network and the time it takes for two machines to communicate with each other, each machine gets a partial view of the global state.

A fundamental criterion in the design of robust distributed systems is to embed the capability of recovery from unforeseen perturbation. While most of the existing systems cater to permanent failures by introducing redundant components, the issue of transient failures is often ignored or inadequately addressed. It is possible to encounter a transient malfunction due to message corruption, sensor malfunction or incorrect read/write memory operations, that transforms the global state of the system into an illegal state, from which recovery is not guaranteed. Examples are token-ring networks in which the token is lost or duplicate tokens are generated, or sliding window protocols in which the window alignment is lost due to transient errors. The essence of these examples is that if the set of possible global states of a distributed system is partitioned into legal and illegal states, then transient failures can potentially put the system into an illegal state, which may continue indefinitely.
unless it is externally detected and suitable corrective measures are taken.

A self-stabilizing system guarantees that regardless of the initial global state, legal or illegal, in which the system started, the system is guaranteed to recover to a legal state in a finite number of steps and remain in the legal configuration thereafter, until a subsequent malfunction occurs. This property makes the system more robust. No startup or initialization procedure needs to be used because the system stabilizes by itself. If one machine fails and restarts, its local state may cause an illegal global state, but the system will correct itself in a finite amount of time. The ability of the system to correct certain errors without outside intervention makes a self-stabilizing system more reliable and more desirable than systems that are not self-stabilizing.

Self-stabilization in distributed system was originally introduced by Dijkstra[4]. Lamport [13] has regarded the original work on self-stabilization by Dijkstra as a milestone in fault-tolerant system designs. The application of self-stabilization has since expanded to many areas of study [5, 8]. Some of the areas are related to distributed systems are communication protocols, load balancing, leader election, deadlock detection, calculation of network metrics, network routing, distributed process control, etc.

The goal of this research is to design fault-tolerant distributed systems using self-stabilization. The first part of this thesis presents a self-stabilizing shortest path algorithm for a distributed network of directed acyclic graph(DAG) topology. The second part presents a self-stabilizing ranking algorithm for a distributed tree network. A distributed system can be easily represented by the above topologies or it could be converted to such a representation as in[11, 10]. Algorithms exist for these problems, but they are not self-stabilizing. The first algorithm is useful for routing considerations in a network. The motivation for the second problem comes from the fact that many distributed file applications like selection are achieved more efficiently in sorted than in arbitrary files.
1.1 Notations

Formally, an algorithm is said to be self-stabilizing if it satisfies the following two properties\[1\].

1. **Closure**: An algorithm is said to be closed if once a legal global state is reached, the system remain in legal global state as long as no perturbation occur in the system.

2. **Convergence**: An algorithm is said to be convergent if the system is not in a global state, then in finite time, the system will move into a legal global state.

The algorithm proposed in this thesis, satisfies both the above properties. The algorithm for each processor is of the form:

```
< statement >
< statement >
...
< statement >
```

Each statement is of the form:

```
< guard > ⇒
< action >
```

A *guard* is a boolean expression over the variables/registers that a processor can read (its own along with those of adjacent processors). If some processor has a statement whose guard is true, then that processor has a *privilege* and may make a *move*, i.e., execute the corresponding *action*.

If several privileges exist in the system, the execution depends on the scheduler which is being used. A central or serial scheduler\[4\] schedules one processor at a time to make a move. A distributed or parallel scheduler\[2, 9\] permits simultaneous moves by the processors. In presence of this scheduler, any subset of the set of privileged processors can move at the same time. The algorithms presented here work in the presence of any of the above schedulers and also make no assumptions about the fairness of the scheduler\[6\].
Chapter 2

SHORTEST PATH ALGORITHM

The shortest path in a directed graph is a classic problem in computer science that can be seen with its sequential solution in any standard algorithms or data structures text such as [7].

This chapter presents a self-stabilizing algorithm for creating a shortest path routing table at each node in a distributed system. The underlying topology of the system under consideration is a directed acyclic graph (DAG). When the system stabilizes, the table at each node provides shortest routes from that node to all reachable nodes in the DAG. This is done by first topologically sorting the successors of each node, and then using this information to create the routing table at each node.

The DAG G consists of n processors (or nodes), denoted as $P_1...P_n$. $P_j$ is a successor to $P_i$, if there is a directed path from $P_i$ to $P_j$. $P_j$ will be called as an immediate successor to $P_i$, iff $<P_i, P_j>$ is an edge in G. If $P_j$ is a successor of $P_i$, then $P_i$ is a predecessor of $P_j$, and in the same way if $P_j$ is an immediate successor to $P_i$, then $P_i$ is an immediate predecessor of $P_j$. The processor with no immediate predecessor is referred to as root and the processors with no immediate successors are referred to as leaves of the network.

Each node can read the values of the registers of its immediate successors, but can write only to its own registers.
The proposed algorithms are self-stabilizing such that the nodes with no successors stabilize first. Then the immediate predecessors to those nodes stabilize, and so on up to those nodes with no predecessors. These algorithms also work for the graphs with more than one root. In that case each root will have the shortest routes to all the processors accessible from that root in its table.

Section 2.1 presents the algorithms in detail, section 2.2 provides the proofs of correctness, and section 2.3 provides the summary and closing remarks concerning these algorithms.

2.1 The Algorithms

2.1.1 The Topological Sort Algorithm

The topological sort of a DAG $G$ is a linear ordering of all its processors such that for any two processors $P_i, P_j$ in $G$, such that if $P_i$ is an immediate predecessor of $P_j$ in the network, then $P_i$ precedes $P_j$ in the linear ordering. In the algorithm proposed in this paper, each processor constructs a partial list of this topologically sorted list of the graph which contains all the processors that are accessible from that processor, i.e. that processor's successors and that processor itself. In other words we can say each processor $P_i$ constructs a topological sort of that part of the graph $G_i$, which is rooted at that processor. In this way when the system is stabilized, we will have the topologically sorted list for the entire graph available at the root.

Let $IS_i$, $S_i$ denote the set of processors which are immediate successors and successors to $P_i$ respectively. The topologically sorted successor set of $P_i$ is maintained in the form of a list, $list_i$. This $list_i$ is an ordered set of processors and it can be visualized as $\{P_i, P_j,...P_k\}$. The ordering of the processors in $list_i$ is represented by their position in $list_i$, i.e., the first processor in $list_i$ is of order 1, the second is of order 2 and so on. It can be seen in the stabilized sorted $list_i$ of $P_i$, $P_i$ itself occupies the first position, i.e., order of 1. We will use the term $order_i(P_j)$ to denote the order of $P_j$ in $list_i$. 
6

\( P_i \) also keeps two variables \( tlist_i \) and \( U_i \) which are of type set. Since \( list_i \) is accessed by the later shortest path algorithm \( P_i \) uses \( tlist_i \) as a temporary list for its computation and once \( P_i \) is done with updating \( tlist_i \) from the information available from its immediate successors, \( P_i \) copies \( tlist_i \) into \( list_i \). To implement this critical section \( P_i \) uses a boolean variable \( copy_i \). \( U_i \) is used to keep track of all the elements already included to it's \( list_i \), through the append operation discussed below.

Now we define some operations on our list data type.

(1) Append operation: This operation \( \text{append}(\text{list}_j, \text{list}_i) \) appends the elements of \( \text{list}_j \) to the front of \( \text{list}_i \). If \( \text{list}_i \) was null before this operation then after this operation \( \text{list}_i \) will just contain only the elements of \( \text{list}_j \).

(2) Set Difference: This operation \( \text{list}_i - \text{list}_j \) represents all the elements of \( \text{list}_i \) but excluding the common elements between \( \text{list}_i \) and \( \text{list}_j \). For example, if \( \text{list}_i \) is 1, 4, 2, 7 and \( \text{list}_j \) is 9, 4, then \( \text{list}_i - \text{list}_j \) is the set 1, 2, 7. Note this set difference operation maintains the order among the elements in \( \text{list}_i \).

(3) Union operation: This is a regular union operation among sets.

When the system is unstable \( \text{list}_i \) may not represent correct and sorted successor set of \( P_i \). It may even include illegal node ids., which are not present in the system. But when the system eventually stabilizes \( \text{list}_i \) of each \( P_i \) represent correct and sorted successor set of \( P_i \). We will denote the correct sorted successor list of \( P_i \) as \( \text{topsort}_i \text{list}_i \). For a leaf node \( \text{topsort}_i \text{list}_i \) contains just itself. For the other nodes \( \text{topsort}_i \text{list}_i \) can be formally defined as:

\[
\text{topsort}_i \text{list}_i \equiv (\forall P_j: P_j \in S_i \land P_j \in \text{list}_i) \land \\
(\forall P_j, P_k: P_j, P_k \in S_i \land P_k \in IS_j \land \text{order}_i(P_j) < \text{order}_i(P_k))
\]

The algorithm for each processor \( P_i \) appears as follows:

(R1) \( IS_i = \emptyset \implies \)

\[
\begin{align*}
&\text{copy}_i := \text{false}; \\
&tlist_i := P_i; \\
&\text{copy}_i := \text{true}
\end{align*}
\]
(R2) $IS_i \neq \emptyset \implies$
\begin{align*}
&\text{copy}_i := \text{false}; \\
&t\text{list}_i := \emptyset; \\
&U_i := \emptyset; \\
&\text{do } \forall P_j: P_j \in IS_i \\
&\quad \text{append}(\text{list}_j - U_i, t\text{list}_i); \\
&\quad U_i := U_i \cup \text{list}_j \\
&\text{enddo} \\
&\text{append}(P_i, t\text{list}_i); \\
&\text{copy}_i := \text{true}
\end{align*}

(R3) $\text{copy}_i = \text{true} \implies$
\begin{align*}
&t\text{list}_i := t\text{list}_i
\end{align*}

The leaf node $P_i$ simply puts itself into its $t\text{list}_i$. The non-leaf node starts its action by initializing it's $t\text{list}_i$ and $U_i$ sets to null. Then it appends each of it's immediate successor's $\text{list}_j$ to it's $t\text{list}_i$. When doing so, it takes care that it does not include an element twice in it's $t\text{list}_i$, i.e., by doing $\text{list}_j - U_i$, $P_i$ makes sure that elements already in $t\text{list}_i$ are not included again when appending it's immediate successor's $\text{list}_j$. This is possible since more than one immediate successor may have an element in common. When $P_i$ is done with all it's immediate successors $\text{list}_j$, its action is complete. All nodes in the system will upon completing the above tasks copy their $t\text{list}_i$ to the permanent $\text{list}_i$.

### 2.1.2 The Shortest Path Algorithm

Once the topological sort is stabilized, a correct shortest path routing table can be created. This algorithm uses the $\text{list}_i$ list to decide which nodes are reachable. The weight associated with a directed edge, $(i, j)$ is denoted by $w(i, j)$ and if there is no directed edge $(i, j)$ then $w(i, j)$ is $\infty$. Each node maintains a list of records $\text{route}_i$. The records each have three elements: $\text{route}_i.\text{dest}$, the destination node $P_j$ of the particular record; $\text{route}_i.\text{next}$, the immediate successor to $P_i$ that lies along
the shortest path to \( P_j \); and \( \text{route}_i.weight \), the weight of the shortest path from \( P_i \) to \( P_j \). We will use the notation \( \text{route}_i.next[j] \) and \( \text{route}_i.weight[j] \) to refer to the immediate successor to \( P_i \) along the shortest path to destination \( P_j \) and the weight of this path respectively. We also need a couple of functions for this algorithm. The function \( \text{min} \) takes the weights of the paths to a successor \( P_j \) from the node under consideration, say \( P_i \) and returns the minimum of the above. The function \( \text{min.id} \) takes this minimum weight as an argument and returns the index of the immediate successor to \( P_i \), which lies in that shortest path to \( P_j \). Also it has to be noted that though \( P_i \) appears in its own \( \text{list}_i \), since there is no self-path to it, the \( \text{route}_i.next[i] \) and \( \text{route}_i.weight[i] \) will be 0 and \( \infty \) respectively.

The algorithm for each \( P_i \) is as follows:

(R1) \( P_j \in \text{list}_i \land P_j \notin \text{route}_i.dest \implies \)

\[
\text{route}_i.dest := \text{route}_i.dest \cup \{P_j\}; \\
\text{route}_i.next[j] := 0; \\
\text{route}_i.weight[j] := \infty
\]

(R2) \( P_j \notin \text{list}_i \land P_j \in \text{route}_i.dest \implies \)

\[
\text{route}_i.dest := \text{route}_i.dest - \{P_j\}
\]

(R3) \( P_j \in IS_i \land P_j \notin \text{list}_k \land P_k \in IS_i \implies \)

\[
\text{route}_i.weight[j] := \text{weight}(i,j); \\
\text{route}_i.next[j] := P_j
\]

(R4) \( P_j \notin IS_i \land P_j \in \text{list}_k \land P_k \in IS_i \implies \)

\[
\text{route}_i.weight[j] := \text{min}(\text{route}_k.weight[j] + \text{weight}(i,k)|P_k \in IS_i); \\
\text{route}_i.next[j] := P_k
\]

(R5) \( P_j \in IS_i \land P_j \in \text{list}_k \land P_k \in IS_i \implies \)

\[
\text{route}_i.weight[j] := \text{min}(\text{min}(\text{route}_k.weight[j] + \text{weight}(i,k)|P_k \in IS_i),\text{weight}(i,j)); \\
\text{route}_i.next[j] := \text{min.id}(\text{min}(\text{min}(\text{route}_k.weight[j] + \text{weight}(i,k)|P_k \in IS_i),\text{weight}(i,j)))
\]
The first two rules stabilize the destinations that $P_i$ can reach by insuring that the $\text{route}_i.\text{dest}$ fields are equal to $\text{list}_i$. The third rule takes the weight of the path to an immediate successor if no other path exists to this node. The fourth rule takes the minimum path from its immediate successors, $P_k$, plus $\text{weight}(i,k)$, to achieve the shortest path if there is no direct link between $P_i$ and its destination $P_j$. The fifth rule is a combination of rules three and four. It basically applies each of three and four, and then takes the minimum from the two.

2.2 Proofs of Correctness

2.2.1 Topological Sort

For the proofs in this section, we introduce the following legal states description:

$$\text{L1} \equiv IS_i = \emptyset \land \text{list}_i = \{P_i\}$$

$$\text{L2} \equiv IS_i \neq \emptyset \land \text{list}_i = \text{topsort}_i\text{list}_i$$

$$\text{G1} \equiv (IS_i = \emptyset \land \text{L1}) \land (IS_i \neq \emptyset \land \text{L2})$$

Lemma 2.1 The topological sort algorithm converges.

Proof: This will be proved using induction on the distance a node is from a leaf node, which is defined as the integer number of links between the node in question and a leaf node.

**Basis:** Leaf nodes (distance is 0) will first use rule one to make $tlist_i$ contain only themselves. Then using rule three this $tlist_i$ is copied to $\text{list}_i$. Thus, state $\text{L1}$ is reached.

**Induction:** Assume, that all nodes $P_k$ that are distance $x$ from a leaf node have $\text{list}_k$ correct, i.e., these nodes are in $\text{L2}$. We will show that a node $P_i$ at distance $x + 1$ will converge. For all $P_k$, $\text{list}_k$ will be copied onto the front of $tlist_i$ without including any node twice (it is important to note that $tlist_i$ starts as empty before
copying begins). Since each $list_k$ is in topological sort order, since they are in $L_2$ and by the way of $tlist_i$ construction, the resulting $tlist_i$ will also be in topological sort order. Using rule three, $tlist_i$ is copied directly to $list_i$ which puts $P_i$ into $L_2$. Therefore, all nodes are in either $L_1$ or $L_2$, thus meeting $G_1$ a global legal state. The algorithm converges.

It is important to note, that while the arguments above appear to assume that a node at $x+1$ will have immediate successors that are all at $x$, this is not necessary for the algorithm to be correct. The nature of a DAG is such that there must be at least one node that has only leaves as immediate successors. A simple inductive argument as above will show that because of this fact the relationships will be linear as in the above arguments. □

**Lemma 2.2** The topological sort algorithm is closed.

**Proof:** Leaves will continually to run rules one and three, thus copying the same $tlist_i$ which contains only $P_i$ into $list_i$. Thus, leaves will remain in $L_1$ unless a perturbation occurs.

Similarly, non-leaf nodes will continually copy the same $tlist_i$ into $list_i$ (the same inductive argument as in 2.1 can be used). Thus, all non-leaf nodes will remain in $L_2$ unless a perturbation occurs.

Thus, the algorithm is closed. □

**Theorem 2.1** This topological sort algorithm is a correct self-stabilizing topological sort algorithm.

**Proof:** The algorithm is closed and convergent by 2.1 and 2.2, and thus self-stabilizing. 2.1 shows the inductive argument that the correct topologically sorted $list_i$ is held at each node. □

### 2.2.2 The Shortest Path Algorithm

For the proofs in this section, the following legal state description will be used:
\[ \text{G2} \equiv \text{G1} \land \forall P_i, P_j, P_k: P_j \in \text{list}_i \land P_k \in IS_i \land \text{route}_i.\text{dest} = \text{list}_i \land \\
\text{route}_i.\text{weight}[j] = \min(\min(\text{route}_k.\text{weight}[j] + \text{weight}(i, k)), \\
\text{weight}(i, j)) \land \\
\text{route}_i.\text{next} = \min.\text{id}(\min(\min(\text{route}_k.\text{weight}[j] + \text{weight}(i, k)), \\
\text{weight}(i, j))) \]

**Lemma 2.3** *The shortest path algorithm converges.*

**Proof:** Induction on the distance of a node from a leaf node will be used.

**Basis:** Rule two will empty \( \text{route}_i.\text{dest} \), thus making \( \text{G2} \) trivially true.

**Induction:** Assume that all nodes at distance \( x \) from a leaf node will be in state \( \text{G2} \). We will show that a node at \( x + 1 \) will converge. Rules one and two will ensure that all and only the elements of \( \text{list}_i \) will also be elements of \( \text{route}_i.\text{dest} \). Rule three will insure all direct paths will be the shortest paths if no other path exists. Rule four will insure that all minimum paths through immediate successors will be the shortest paths if no direct path exists. Rule five will insure that the minimum of the paths of rules three and four is used if both a direct and indirect path to the destination exists. These rules, three through five, will also insure that either the destination or the successor whose path is used will be the next node in the routing list. Thus, all of the conjuncts of \( \text{G2} \) will be true, and the node will be in state \( \text{G2} \). All nodes will thus converge to \( \text{G2} \). Thus the algorithm converges. \( \square \)

**Lemma 2.4** *The shortest path routing algorithm is closed.*

**Proof:** Once a node is in state \( \text{G2} \), \( \text{route}_i.\text{dest} = \text{list}_i \). Therefore, no privilege exists for rules one and two. Also, \( \text{route}_i.\text{weight} = \min(\min(\text{route}_k.\text{weight}[j] + \text{weight}(i, k)), \text{weight}(i, j)) \) so that no privilege exists for rules three through five. When no privileges exist in the system, no information is changed, and the system remains in \( \text{G2} \). The algorithm is closed. \( \square \)

**Theorem 2.2** *The shortest path routing algorithm is a correct self-stabilizing shortest path routing algorithm for a DAG.*
Proof: The algorithm is closed and convergent by 2.3 and 2.4, and thus a self-stabilizing algorithm. Since any sub-path of a shortest path is itself a shortest path, the \( \min(\min(\text{route}_k.\text{weight}[j] + \text{weight}(i, k)|P_k \in IS_i), \text{weight}(i, j)) \) will be the shortest path \( P_i \) holds. \( P_k \) holds the correct shortest path using a similar inductive argument to that in 2.3. \( \square \)

2.3 Summary

Two algorithms have been presented in this chapter that combined create topologically sorted list of the nodes reachable from each node, and the corresponding shortest path routing table to reach those nodes.

There are modifications that can be trivially noted. First, the two algorithms could be combined such that the topological sort would create the sorted list directly in the \( \text{route}_i.\text{dest} \) field of the \( \text{route}_i \) array, and thus the first two rules of the second algorithm could be eliminated. Second, to achieve the shortest path routing table, the sorted list is not necessary. Merely a list of all reachable nodes needs to be collected. However, it must be noted that the topological information of a network can be useful for many other reasons, so \( \text{list}_i \)'s usefulness is not contained to the shortest path problem only.
Chapter 3

RANKING ALGORITHM

The ranking and associated sorting and selection in a distributed network are well studied. Ranking means that the numbers 1, 2,..., N have to be assigned to the processors according to their values. Distributed algorithms exist[14] for the above problem, but they are not self-stabilizing and they require that a non-empty subset of the processors start the algorithm. This chapter presents a distributed self-stabilizing ranking algorithm for a network of N processors arranged in the form a tree.

Section 3.1 presents the algorithm in detail, section 3.2 provides the proofs of correctness and section 3.3 provides the summary of this chapter.

3.1 The Ranking Algorithm

3.1.1 Description of the protocol

Each processor $i$ has a value denoted by $i.val$ and an unique id. denoted by $i.id$. These values need not be distinct. The values are positive integers and the range for the processors' ids. is between 1 and N, where N, the number of nodes in the tree network is known to all the nodes. The ranking algorithm proposed here, determines the rank of node $i$ denoted by $i.rank$ based on its value, i.e., the node with the minimum value
is assigned the rank 1 and the maximum value is assigned the rank N. In case of a
tie between two values, the ids. of the nodes is used as a secondary criteria in the
comparison, with the result that the node with lesser id. will have the lesser rank.

Each processor \( i \) can perform read/write operations on its local registers, but it
can only read from registers of its neighbors, i.e., its parent and children. Processor
\( i \) has the id. of its parent in the register \( i.parent \) and has the ids. of its children in
the register \( i.child \) which is of the type set. we assume that an underlying spanning
tree protocols as in[11, 10] maintains the consistency of these fields. The unique
node in the network with no parent is referred as the root and the leaf nodes are
characterized by \( i.child = \emptyset \). Each processor \( i \) also maintains a two tuple register
\( <winner_.val, winner_.id> \) and another register \( i.phase \). The purpose of these regis-
ters will be clear as we explain the algorithm.

The execution of the protocol by each node is done in cycles, i.e., when the system
completes a cycle it starts a new cycle. Each cycle has two stages. The first stage does
the pre-processing for the second stage. The second stage does the actual ranking.

The pre-processing stage does the job of placing at each node \( i \)'s \( <winner_.val, win-
ner_.id> \) tuple, the minimum \( <val, id> \) pair in the sub-tree rooted at \( i \). In this stage,
the leaf nodes, as well a node which finds its own \( <val, id> \) pair as the minimum
among the nodes in the subtree rooted at it, places its own \( <val, id> \) pair in its
\( <winner_.val, winner_.id> \) tuple. So is the case the for the node \( i \) with minimum
\( <val, id> \) pair. The parent \( j \) of \( i \) finds that \( i \)'s \( <val, id> \) pair as the minimum among
itself and its children, and hence places \( i \)'s \( <val, id> \) pair as the winner in its tuple
\( <winner_.val, winner_.id> \). The parent of \( j \) also does the same. Thus the node \( i \)'s
\( <val, id> \) pair rises all the way up to the the root through the nodes in the path
between \( i \) and the root, meaning all the nodes in the path between \( i \) and the root
have the same value, i.e., \( i \)'s \( <val, id> \) for their respective \( <winner_.val, winner_.id> \)
tuple. So at the end of this stage, the root will have the minimum \( <val, id> \) pair of
the tree network in its \( <winner_.val, winner_.id> \) tuple.

Now the next ranking stage starts. The ranking stage consists N phases to deter-
mine the N ranks in the system. At the start of the phase one, since the root already
has the minimum \(<val, id>\) pair in its \(<winner.val, winner.id>\) tuple, all it has to do is to send the first rank to \(i\). To enable this the root sets its phase register to 1. This rank is passed all the way down to \(i\) by the same nodes in the path between root and \(i\) through which \(i\)'s \(<val, id>\) pair raised to the root. The above rank passing is done by those nodes, by each node copying the rank in it's parent's phase to its own phase register. Once the rank reaches the node \(i\), it takes that rank by setting its \(i.rank\) with this rank. Now we have determined the first rank in the network. Also since \(i\) has got it's rank it is not going to compete any further in this ranking stage. Now \(i\) replaces its \(<winner_i.val, winner_i.id>\) tuple with the second minimum \(<val, id>\) pair in the subtree rooted at \(i\) if there is one, else with the empty tuple \(<0, 0>\). It has to be noted that this second minimum \(<val, id>\) pair in the subtree rooted at \(i\), may not be the second minimum \(<val, id>\) pair of the tree network. Once \(i\) is done with assigning the next minimum \(<val, id>\) pair to it's \(<winner_i.val, winner_i.id>\) tuple, it's parent selects the minimum \(<val, id>\) pair among itself and it's children and assigns this \(<val, id>\) pair to its \(<winner.val, winner.id>\) tuple. Then the parent of this parent does the same action and so on, until finally the root does the same action of selecting it's next minimum \(<val, id>\) pair for it's \(<winner.val, winner.id>\) tuple.

Thus in each \(i^{th}\) phase, the root determines the \(i^{th}\) ranked node, sends down this rank to the concerned node and in turn collects the next minimum \(<val, id>\) pair in the system, for the \((i + 1)^{th}\) rank.

This goes on until all the \(N\) nodes are ranked, i.e., for \(N\) phases. When the root sends down the \(N^{th}\) rank, it gets back the \(<0, 0>\) tuple as the next minimum \(<val, id>\) pair, indicating all the nodes are ranked and there is no further node in the competition.

At this stage root applies R1 and start the pre-processing phase.

The special node root maintains in addition to the above registers, the register \(temp\) to keep track of the phase.

Under normal circumstances, i.e., when there is no perturbation in the system, the system just cycles through pre-processing stage and ranking stage. But when there
is perturbation, the system will be in an illegal state. So to bring the system back to normal, we introduce an error correction part to the protocol. The error correction part brings the system to pre-processing stage and from then on the regular cycle continues.

The error correction strategy is as follows: In the ranking stage, a node $i$'s $i$.phase register should be normally zero. When $i$ has to pass down a rank sent by the root to the nodes below it, it copies its parent's phase register into its $i$.phase. So, $i$.phase can have a value in the range between 1 and $N$, i.e., $i$.phase will be $k$ in the $k$'th phase in the above situation. In addition $i$.phase can take a value of -1, when it passes down the signal sent by the root that the pre-processing stage of a new cycle has started. The $<\text{winner}.val, \text{winner}.id>$ tuple of node $i$ is always supposed to have the minimum $<\text{val}, \text{id}>$ pair of the subtree rooted at $i$ except when a new $<\text{val}, \text{id}>$ pair rises up in return to a rank sent down by $i$ and when it passes down the pre-processing initiation message of the root during which $<\text{winner}.val, \text{winner}.id>$ is equal to $<0,0>$. When node $i$ is not in any of the above state, it is in an illegal state.

Since we have only $N$ phases to rank the $N$ nodes in the tree network, any value greater than $N$, for $i$.phase is an illegal value, which we will use it to our advantage. So when $i$ is in an illegal state, we will use the value $N+1$ for $i$.phase to indicate that $i$ is in error. If a node is in error, the parent of that node also goes into error state by setting its phase to $N+1$. This goes all the way up to the root, and when root finds that it's phase is more than $N$, it starts the pre-processing stage of a new cycle and thus bringing the system back to normal.

The registers term of the root and $i$.rank of any node $i$ doesn't pose any problem in the stabilization. Even if they are in illegal state, the root will eventually have either its $<\text{winner}.val, \text{winner}.id>$ equal to $<0,0>$ without its phase becoming $N$ or its phase $>N$ without its $<\text{winner}.val, \text{winner}.id>$ tuple becoming $<0,0>$, thus triggering the pre-processing stage of a new cycle.

Finally, before presenting the algorithm, we will define the procedure $\text{find}\_\text{min}()$ used by the algorithm of node $i$ as follows:

Procedure $\text{find}\_\text{min}()$:
\[ k : k.\text{parent} = i \land (i.\text{rank} = 0) \Rightarrow \]
\[ \text{return}(\text{min}(<i.\text{val}, i.\text{id}>, <\text{winner}_k.\text{val}, \text{winner}_k.\text{id}>)) \]
\[ k : k.\text{parent} = i \land (i.\text{rank} \neq 0) \Rightarrow \]
\[ \text{return}(\text{min}(<\text{winner}_k.\text{val}, \text{winner}_k.\text{id}>)). \]

Basically by this procedure \( i \) returns the minimum \(<\text{val}, \text{id}>\) pair among \( i \)'s own \(<i.\text{val}, i.\text{id}>\) tuple and the \(<\text{winner}.\text{val}, \text{winner}.\text{id}>\) tuple of it's children, if \( i \) is not ranked. Otherwise \( i \) returns the minimum \(<\text{val}, \text{id}>\) pair among the \(<\text{winner}.\text{val}, \text{winner}.\text{id}>\) tuple of it's children. In case \( i \) is ranked and all it's children have their respective \(<\text{winner}.\text{val}, \text{winner}.\text{id}>\) tuple as \(<0,0>\), then \( i \) returns \(<0,0>\). If the procedure finds more than one minimum \(<\text{val}, \text{id}>\) pair, it returns that tuple.

### 3.1.2 Pre-processing stage

Rule R1 is for the root to initiate the pre-processing stage of a new cycle. This signal is passed down the tree to the leaves by R2. This process is referred as the broadcasting of pre-processing initiation signal. When the leaves get this signal they set their their respective \(<\text{winner}.\text{val}, \text{winner}.\text{id}>\) tuple with their own \(<\text{val}, \text{id}>\) pair and \text{phase} to 0, by R3, thus reflecting the broadcast wave and starting a converge-cast wave towards the root. This converge-cast done by means of executing R4, enables the nodes other than the leaves to collect the minimum \(<\text{val}, \text{id}>\) pair of the subtree rooted at them, in their respective \(<\text{winner}.\text{val}, \text{winner}.\text{id}>\) tuple.

\[(R1) \; i = \text{root} \wedge \]
\[(i.\text{phase} > N \vee <\text{winner}_i.\text{val}, \text{winner}_i.\text{id}> = <0,0>) \Rightarrow \]
\[i.\text{phase} := -1;\]
\[<\text{winner}_i.\text{val}, \text{winner}_i.\text{id}> := <0,0> ;\]
\[i.\text{rank} := 0;\]
\[\text{temp} := 0\]
(R2) \( (j = i.\text{parent} \And j.\text{phase} = -1 \And \langle \text{winner}_j.\text{val}, \text{winner}_j.\text{id}\rangle = <0,0> \And j.\text{rank} = 0) \And \\
\neg (i.\text{phase} = -1 \And \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle = <0,0> \And i.\text{rank} = 0) \And \\
\neg (i.\text{phase} = 0 \And \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle = \text{find}_\text{min}() \And i.\text{rank} = 0) \implies \\
i.\text{phase} := -1; \\
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle := <0,0>; \\
i.\text{rank} := 0

(R3) \( i.\text{child} = \emptyset \And \\
(i.\text{phase} = -1 \And \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle = <0,0> \And i.\text{rank} = 0) \implies \\
i.\text{phase} := 0; \\
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle := \text{find}_\text{min}()

(R4) \( i.\text{phase} = -1 \And \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle = <0,0> \And i.\text{rank} = 0) \And \\
(\forall k: k \in i.\text{child} \And k.\text{phase} = 0 \And \langle \text{winner}_k.\text{val}, \text{winner}_k.\text{id}\rangle \neq <0,0> \And \\
k.\text{rank} = 0) \implies \\
i.\text{phase} := 0; \\
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id}\rangle := \text{find}_\text{min}()

3.1.3 Ranking stage

The root initiates a phase in the ranking stage by executing R5, i.e., it increments temp and sets its phase with this value. By R6, the root collects the next minimum \( <\text{val}, \text{id}> \) pair for the next phase, which was returned in reply to the rank sent down by the root by executing R5. The intermediate nodes send up the the next minimum \( <\text{val}, \text{id}> \) pair to the root in reply to a rank sent down through these nodes earlier in that phase, by executing R7. The rank determined by the root for that phase is sent down the tree by R8 and R9. The intermediate nodes send down the rank by setting their phase to as of their parent through R8. R9 is same as R8, except that it is for the leaf nodes. Rules R10(applicable for root), R11(for intermediate nodes) and R12(for leaf nodes) are for that particular special node which won that phase. This node take the rank determined by that phase and sent up the next minimum
<val, id> pair of its subtree in reply to the rank, by executing R10 or R11 or R12 as applicable to it.

(R5) \( i = \text{root} \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq <0,0> \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> = \text{find\_min}() \land \)
\( i.\text{phase} = 0 \land 0 \leq \text{temp} \leq N \land \)
\( (\forall k: k \in i.\text{child} \land k.\text{phase} = 0) \implies \)
\( \text{temp} := \text{temp} + 1; \)
\( i.\text{phase} := \text{temp} \)

(R6) \( i = \text{root} \land <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq \text{find\_min}() \land \)
\( 1 \leq i.\text{phase} \leq N \land i.\text{phase} = \text{temp} \land \)
\( (\forall k: k \in i.\text{child} \land k.\text{phase} = 0) \implies \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> := \text{find\_min}(); \)
\( i.\text{phase} := 0 \)

(R7) \( i \neq \text{root} \land i.\text{child} \neq \emptyset \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq \text{find\_min}() \land 1 \leq i.\text{phase} \leq N \land \)
\( (j = i.\text{parent} \land i.\text{phase} = j.\text{phase} \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> = <\text{winner}_j, \text{val}, \text{winner}_j, \text{id}> \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq <0,0> \land \)
\( (\forall k: k \in i.\text{child} \land k.\text{phase} = 0) \implies \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> := \text{find\_min}(); \)
\( i.\text{phase} := 0 \)

(R8) \( i \neq \text{root} \land i.\text{child} \neq \emptyset \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> = \text{find\_min}() \land i.\text{phase} = 0 \land \)
\( (\forall k: k \in i.\text{child} \land k.\text{phase} = 0) \land \)
\( (j = i.\text{parent} \land 1 \leq j.\text{phase} \leq N \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> = <\text{winner}_j, \text{val}, \text{winner}_j, \text{id}> \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq <0,0> \land \)
\( <\text{winner}_i, \text{val}, \text{winner}_i, \text{id}> \neq <0,0> \land \)
\( \implies \)
\( i.\text{phase} := j.\text{phase} \)
(R9) $i.\text{child} = \emptyset \land$

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \text{find\_min()} \land i.\text{phase} = 0 \land
\]

\[
(j = i.\text{parent} \land 1 \leq j.\text{phase} \leq N \land
\]

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle \text{winner}_j.\text{val}, \text{winner}_j.\text{id} \rangle \land
\]

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle \neq \langle 0, 0 \rangle \implies
\]

\[
i.\text{phase} := j.\text{phase}
\]

(R10) $i = \text{root} \land$

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle i.\text{val}, i.\text{id} \rangle \land
\]

\[
1 \leq i.\text{phase} \leq N \land i.\text{phase} = \text{temp} \land
\]

\[
(\forall k : k \in i.\text{child} \land k.\text{phase} = 0) \implies
\]

\[
i.\text{rank} := i.\text{phase};
\]

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle := \text{find\_min();}
\]

\[
i.\text{phase} := 0
\]

(R11) $i \neq \text{root} \land i.\text{child} \neq \emptyset \land$

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle i.\text{val}, i.\text{id} \rangle \land 1 \leq i.\text{phase} \leq N \land
\]

\[
(j = i.\text{parent} \land \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle \text{winner}_j.\text{val}, \text{winner}_j.\text{id} \rangle \land
\]

\[
i.\text{phase} = j.\text{phase}) \land
\]

\[
(\forall k : k \in i.\text{child} \land k.\text{phase} = 0) \implies
\]

\[
i.\text{rank} := i.\text{phase};
\]

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle := \text{find\_min();}
\]

\[
i.\text{phase} := 0
\]

(R12) $i.\text{child} = \emptyset \land$

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle i.\text{val}, i.\text{id} \rangle \land 1 \leq i.\text{phase} \leq N \land
\]

\[
(j = i.\text{parent} \land \langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle = \langle \text{winner}_j.\text{val}, \text{winner}_j.\text{id} \rangle \land
\]

\[
i.\text{phase} = j.\text{phase}) \implies
\]

\[
i.\text{rank} := i.\text{phase};
\]

\[
\langle \text{winner}_i.\text{val}, \text{winner}_i.\text{id} \rangle := \text{find\_min();}
\]

\[
i.\text{phase} := 0
\]
3.1.4 Error correction

The rules R13, R14 and R15 are applicable respectively to the root, the intermediate nodes and the leaves. Whenever these nodes find that they are in error and their respective phase register is not greater N, they set their phase register to N+1, by executing R13 or R14 or R15 as applicable to it. R16 is used to transfer this information to the root, so that the root can start the pre-processing stage of a new cycle.

(R13) \( i = root \land \)
\[ (i.phase < -1 \lor \]
\[ (i.phase = -1 \land \]
\[ \neg(<winner_i.val, winner_i.id> = <0,0> \land i.rank = 0 \land temp = 0) \lor \]
\[ (1 \leq i.phase \leq N \land \neg(i.phase = temp)) \lor \]
\[ \neg(0 \leq temp \leq N) \lor \]
\[ (<winner_i.val, winner_i.id> \neq find.min() \land \]
\[ \neg(1 \leq i.phase \leq N \land i.phase = temp)) \land \]
\[ i.phase \neq N \implies \]
\[ i.phase := N + 1 \]

(R14) \( i \neq root \land i.child \neq \emptyset \land \)
\[ (i.phase < -1 \lor \]
\[ (i.phase = -1 \land \]
\[ \neg(<winner_i.val, winner_i.id> = <0,0> \land i.rank = 0) \land \]
\[ (j = i.parent \land \]
\[ \neg(j.phase = -1 \land \]
\[ <winner_j.val, winner_j.id> = <0,0> \land j.rank = 0)) \lor \]
\[ (1 \leq i.phase \leq N \land \]
\[ (j = i.parent \land \]
\[ \neg(<winner_i.val, winner_i.id> = <winner_j.val, winner_j.id> \land \]
\[ i.phase = j.phase)) \lor \]
\[ (<winner_i.val, winner_i.id> \neq find.min() \land \]
\[ (j = i.parent) \]
\[ \land \neg(<\text{winner}_i.val,\text{winner}_i.id> = <\text{winner}_j.val,\text{winner}_j.id> \land 1 \leq j.\text{phase} \leq N \land i.\text{phase} = j.\text{phase}) ) \land \\
( i.\text{phase} \not\in N \implies \\
i.\text{phase} := N + 1 \]

(R15) \[ i.\text{child} = \emptyset \land \\
( i.\text{phase} < -1 \lor \\
( i.\text{phase} = -1 \land \neg( <\text{winner}_i.val,\text{winner}_i.id> = <0,0> \land i.\text{rank} = 0) \land \\
( j = i.\text{parent} \land \\
\neg( j.\text{phase} = -1 \land <\text{winner}_j.val,\text{winner}_j.id> = <0,0> \land j.\text{rank} = 0)) ) ) \lor \\
(1 \leq i.\text{phase} \leq N \land \\
( j = i.\text{parent} \land \\
\neg( <\text{winner}_i.val,\text{winner}_i.id> = <\text{winner}_j.val,\text{winner}_j.id> \land \\
i.\text{phase} = j.\text{phase}) ) ) \lor \\
<\text{winner}_i.val,\text{winner}_i.id> \neq \text{find.min}() \land \\
i.\text{phase} \not\in N \implies \\
i.\text{phase} := N + 1 \]

(R16) \[ (\exists k : k \in i.\text{child} \land k.\text{phase} > N) \land \\
\neg( i.\text{phase} = -1 \land <\text{winner}_i.val,\text{winner}_i.id> = <0,0> \land i.\text{rank} = 0) \land \\
i.\text{phase} \not\in N \implies \\
i.\text{phase} := N + 1 \]

### 3.2 Proof of correctness

Before beginning the proof, we formally define a specific legitimate state LS. The N nodes in the system will be correctly ranked by the ranking stage, once the system reaches state LS. All the states reachable from LS are also legitimate states meaning the protocol is closed. LS is defined as:

\[ LS \equiv (\forall i: <\text{winner}_i.val,\text{winner}_i.id> = \text{find.min()} \land i.\text{phase} = 0 \land \\
i.\text{rank} = 0) \land (\text{temp} = 0) \]
This state represents the beginning of the ranking state established by the end of the pre-processing stage. The root can now apply R5 to initiate the first phase of the N-phase ranking stage. Even though the protocol will work with an unfair scheduler, for simplicity of the proof, we will assume a fair scheduler for the following discussion.

Starting from any arbitrary state, the system reaches state LS through the following states:

1. If a node is in error it will set its phase to N+1 by any one of the corresponding rules R13 or R14 or R15.
2. The above error condition is propagated up to the root by the nodes in between this node and the root, by making their respective phase registers to N+1 by R16.
3. Once the root is in error state, i.e., root's phase > N, it starts the pre-processing stage of a new cycle by applying R1.
4. End of pre-processing brings the system to LS.

Thus the system converges to LS from any arbitrary state in finite time. It is obvious, that once the system reaches the state (2) mentioned above, it will cycle through pre-processing and ranking stages.

**Lemma 3.1** When the system is in an illegal state, it will be able to recover itself and bring the system to the beginning of pre-processing stage.

**Proof:** When a node is in error, it doesn't have any rule of ranking stage applicable to it. The only rule applicable to it under error conditions is R13(for root) or R14(for intermediate nodes) or R15(leaf nodes), by which the node sets its phase to N+1, if its phase is not already greater than N. A node in an illegal state in the pre-processing stage(after the pre-processing broadcast wave has passed it to the nodes below in its sub-tree, but before the converge cast wave pass through it) doesn't create any problem, since it will set its phase to -1 as its parents phase is still -1.

The parent \(i\) of this node, on seeing its child's phase greater than N, sets its own phase to N+1 by R16. The only time \(i\) will not set its phase to N+1 is, when \(i\) is already engulfed in the downward broadcasting wave of the pre-processing stage, i.e.,
\( i.\text{phase} = -1 \land <\text{winner}_i.\text{val}, \text{winner}_i.\text{id}> = <0,0> \land i.\text{rank} = 0 \) or \( i.\text{phase} \) is already greater than \( N \). If the system is not already in the pre-processing stage, we can inductively prove that all the nodes including the root, in the path from that node in error to the root will set their respective phase registers to \( N+1 \) by \( R16 \).

Once the root has its phase > \( N \), the only applicable to it is \( R1 \). So the root applies \( R1 \) and starts the pre-processing stage of a new cycle.

**Lemma 3.2** *The proposed protocol converges to LS.*

**Proof:** By lemma 3.1, the system comes back to the beginning of pre-processing state, from any arbitrary state in finite time as we have finite number of nodes in the network. It is easy to see that the pre-processing stage takes the system to the beginning of ranking stage, i.e., to LS. Thus the system converges.

**Lemma 3.3** *The proposed protocol is closed.*

**Proof:** Once the system is in state LS, it cycles through ranking and pre-processing stages and hence all the states reachable from LS are also legal states. So the system stays in legal states, once it is in LS. Thus the protocol is closed.

**Theorem 3.1** *The proposed protocol is self-stabilizing.*

**proof:** The protocol is convergent and closed by lemmas 3.2 and 3.3 respectively. Thus the proposed protocol is self-stabilizing.

### 3.3 Summary

This chapter presented a distributed self-stabilizing algorithm for ranking \( N \) processors in a distributed network arranged in the form of a tree, based on their values. The proposed algorithm doesn't need any initialization i.e., the system can be started in any state and it is self-stabilizing.
Chapter 4

CONCLUSIONS

Self-stabilization is an evolving paradigm in fault-tolerant computing. A system is self-stabilizing, if it can automatically recover from transient errors and perturbation. This ability to recover spontaneously from an arbitrary illegal state due to transient errors, makes the system more robust and fault tolerant.

Dijkstra originally introduced the property of self-stabilization in distributed systems by developing three self-stabilizing mutual exclusion algorithms. After this, self-stabilizing was researched a great deal. Now, the study of self-stabilization has expanded to other areas of distributed systems like communication protocols, and other network algorithms, load balancing, clock synchronization, Byzantine generals problem, consensus and commit, and other fault-tolerance problems. Self-stabilization can be used in any area which has well defined global states. It can even be used in areas such as machine learning and neural networks where the legal states in the system are a set of facts.

The goal of our research was to design fault-tolerant distributed systems. In order to achieve this goal, we used self-stabilization.

Chapter 2 presented a self-stabilizing topological sort algorithm and a self-stabilizing shortest path algorithm for a distributed network of DAG topology. The shortest path algorithm used the results provided by the topological sort algorithm. The topological
sort algorithm on its own can be applied to any situation that requires a total ordering to be constructed from the given partial ordering. The shortest path algorithm finds its use in network routing.

Chapter 3 presented a self-stabilizing ranking algorithm for a distributed tree network. The ranking algorithm is a dynamic algorithm. It allows nodes to change their values and the algorithm determines the individual rank of the nodes based on their new values. With minor modifications, this algorithm can be used for selection problems. Also, the algorithm can be extended to sort the values such that the minimum value moves to the node with the minimum id, the next minimum value to the node with the next minimum id and so on. It will be more challenging to rank the values if each node in the network has more than one value.

These self-stabilizing systems require no initialization and they automatically tolerates transient errors (memory crash, message corruption, etc.). These algorithms eliminate the need of manual intervention to restore the system to normalcy from the transient faults and thus avoid the otherwise huge dollar cost in vast decentralized networks. Thus these algorithms improve confidence in the system's data integrity and in turn extends the system usefulness.
Bibliography


