Self-stabilizing tree algorithms

Visalakshi Thiagarajan
University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/559

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 TREE ALGORITHMS

by

Visalakshi Thiagarajan

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
December 1995
The thesis of Visalakshi Thiagarajan for the degree of Master of Science in Computer Science is approved.

Chairperson. Ajoy Kumar Datta, Ph.D

Examinining Committee Member. John Minor, Ph.D

Examinining Committee Member. Roy Ogawa, Ph.D

Graduate Faculty Representative. Ashok Iyer, Ph.D

Interim Graduate Dean. Cheryl L. Bowles, Ed.D

University of Nevada, Las Vegas
December 1995
ABSTRACT

Designers of distributed algorithms have to contend with the problem of making the algorithms tolerant to several forms of coordination loss, primarily faulty initialization. The processes in a distributed system do not share a global memory and can only get a partial view of the global state. Transient failures in one part of the system may go unnoticed in other parts and thus cause the system to go into an illegal state. If the system were self-stabilizing, however, it is guaranteed that it will return to a legal state after a finite number of state transitions. This thesis presents and proves self-stabilizing algorithms for calculating tree metrics and for achieving mutual exclusion on a tree structured distributed system.
3.2 Description of the Protocol .............................................. 23
  3.2.1 System Model .................................................. 23
  3.2.2 The Algorithm .............................................. 24
3.3 Proof of Correctness ..................................................... 25
3.4 Conclusions ............................................................. 28

4 CONCLUSIONS ......................................................... 30

BIBLIOGRAPHY ........................................................ 31
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Ajoy Datta, for his guidance and help, not only in my thesis work but also in several aspects of my work at UNLV. Thanks for being there for me, Dr. Datta.

Thanks to Dr. Roy Ogawa, my friend, philosopher, and guide. Dr. John Minor, thanks for your stimulating lectures. To my department, thanks for the assistantship and other forms of support.

I had a great time at UNLV!

Visalakshi Thiagarajan
Chapter 1

INTRODUCTION

1.1 Self-Stabilization as a Unified Paradigm for Fault Tolerance

A fundamental criterion in the design of robust systems is to embed the capability of recovery from unforeseen perturbances. While most of the existing systems recover from permanent failures by introducing redundant components, the issue of transient failures is often ignored or inadequately addressed. Consider the computation in a distributed system to be a totally or partially ordered sequence of states in state space. It is conceivable to encounter a transient malfunction due to message corruption, sensor malfunction or incorrect read/write memory operations. This would transform the global state of the system into an illegal state from which recovery is not guaranteed. Examples are token-ring networks where the token is lost or duplicated, sliding window protocols where 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 in an illegal state. The system may continue to remain
in an illegal state unless it is externally detected and suitable corrective measures are undertaken.

A self-stabilizing system would, however, recover to a legal configuration in a finite number of steps, regardless of the current state. It would also remain in the legal configuration thereafter, unless another malfunction occurs. This property makes the system more robust. No startup or initialization procedures are needed. Also, if one machine fails and restarts, the system may go into an illegal global state momentarily, but this is corrected in a finite amount of time. The ability of a self-stabilizing system to correct certain errors without outside intervention makes it more reliable and definitely more desirable than others. Thus self-stabilization is an exercise in global convergence through local actions and is a unified model for fault tolerance [17].

1.2 Brief History of Self-Stabilization

The notion of self-stabilization has been prevalent in the field of mathematics and control theory for many years. In the field of distributed systems, the study of self-stabilization was pioneered by Dijkstra who solved the mutual exclusion problem for a ring of processors using this technique. Dijkstra's notion of self-stabilization [9] which originally had a very narrow scope of application is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems. However, Dijkstra did not address the significance of the property of self-stabilization [17]. This fact was belabored by Lamport who said in his address in 1983, at the 3rd ACM Symposium on Principles of Distributed Computing:

I regard this as Dijkstra's most brilliant work - at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance.

The application of self-stabilization has since expanded to many areas of study related to distributed systems: message passing protocols [1, 4, 16], leader election, network
routing, graph algorithms, etc. [3, 7, 10, 13]. These areas are mentioned in [17]. The study of self-stabilization has been formalized in [2], where, an algorithm is defined to be self-stabilizing if it satisfies the following two properties:

(1) **Closure**: An algorithm is said to be closed if, once the system reaches a legal global state, it is guaranteed to remain in a legal state as long as no perturbation occurs.

(2) **Convergence**: An algorithm is said to be convergent if the system will achieve a legal global state in finite time from an illegal state.

### 1.3 Problems with the Self-Stabilization Model

Self-stabilization appears to be easier than other fault tolerant models. For instance, every process is guaranteed to participate in the algorithm and to execute only its code under all circumstances. This differs from, for example, Byzantine failure where some of the processes can actually ignore the code taking arbitrary and even malicious steps in the system. Thus in the self-stabilization model, the 'program' is assumed to be inviolable. Another problem with self-stabilization is that the processes have no way of distinguishing when the system has stabilized. No process can rely on its local variables and counters since processes can be started with arbitrary values. Also, self-stabilizing algorithms can never turn over control to non-stabilizing algorithms since that would require that a process be able to know when the system is stabilized.

### 1.4 Introduction to This Thesis

This research work presents self-stabilizing algorithms for calculating metrics such as diameter, centroid and median and for achieving mutual exclusion on a tree-based system. To calculate the metrics of a general graph, a spanning tree of the graph may be first constructed by means of a self-stabilizing algorithm such as [7] and then the given algorithm may be run. Such a layering of more than one algorithm is used
frequently in self-stabilizing algorithms such as in [10].

Throughout this work, the following shared memory model of a distributed system is used. The model assumes that there are \( n \) nodes \( 1, \ldots, n \) arranged in a tree configuration, one being the root. The tree may be one maintained by a spanning tree protocol over a graph, thus making the model more general. Each node in the network, \( i \), maintains a read/write register \( r_i \) containing several fields.

A state of the system is defined by a value for every field of the registers maintained by the nodes. Each node in the system executes a protocol which has the form

\[
\{\text{Phase name}\} < \text{phase} > \{\text{Phase name}\} \ldots \{\text{Phase name}\} < \text{phase} >
\]

Each phase is of the form

\[
< \text{rule} > \mid \ldots \mid < \text{rule} >
\]

Each rule has the form

\[
< \text{guard} > \longrightarrow < \text{assignment statement} >
\]

A guard is a Boolean expression over the state of a node and its neighbors. An assignment statement updates the state of a node. An rule whose guard is true at some state of the system is said to be enabled at that state.

The read/write register \( r_i \) of node \( i \) contains several fields depending on the protocol it executes. \( i \) can perform read/write operations on its local register \( r_i \), but it can only read from registers \( r_j \) of its neighbors (i.e., its parent and children).

The rest of the thesis is organized as follows. Chapter 2 presents a set of rules for calculating the diameter of a tree-based distributed system and for identifying its centroids and medians. The proof of correctness and complexity analysis are then presented. In Chapter 3, an algorithm to achieve mutual exclusion among a set of processors connected in a tree configuration is presented along with the proof. Chapter 4 summarizes the thesis and elaborates on future research possibilities.
Chapter 2

TREE METRICS

2.1 Introduction

Topological information, such as location of centroid and median, plays an important role in distributed networks. This information is used for dynamic routing of messages between nodes. But it cannot be taken into account once and for all at design time since several unpredictable factors make it time varying. The problem of dynamically finding the diameter and locating centroids and medians of a tree structured network therefore assumes importance. This chapter presents protocols for finding the diameter and locating centroids and medians of a dynamic tree network. The solutions presented require only local topological knowledge at each node, and are self-stabilizing [9, 11, 17]. The self-stabilizing algorithm terminates after it computes the metrics, but any unexpected perturbation reactivates it, and possibly new values for the metrics are computed if there are changes in the network topology. Work has been done by Karaata et al [14] in this area. They require that each action have a very large atomicity whereas we have no such requirement. Also, every node in the network knows the identity of the centroid and median of the network when our protocol terminates, thus making it an ideal underlying protocol for routing purposes.
In [14] only the medians and the centroids themselves *know* who they are.

The rest of the chapter is organized as follows. Section 2.2 contains a description of the protocols while Section 2.3 provides proofs of correctness. Section 2.4 states some conclusions.

### 2.2 Description of the Protocol

The read/write register \( r_i \) of node \( i \) contains the following fields:

- \( r_i.parent \): has the node index of the parent of \( i \), except for the root which has zero
- \( r_i.ht \): contains the height of \( i \)
- \( r_i.dt.up \): used for convergecast of diameter information
- \( r_i.dt.down \): final result of the diameter of the tree
- \( r_i.center.up \): used for convergecast of centroid information
- \( r_i.center.down \): final result of centroid of the tree
- \( r_i.count \): number of nodes in the subtree rooted at \( i \)
- \( r_i.nodes \): total number of nodes in the tree
- \( r_i.median.up \): used for convergecast of median information
- \( r_i.median.down \): final result of median of the tree

Node \( i \) depends on node \( j \) if a change in the state of \( j \) enables some rule of \( i \).

A phase is defined to be *convergent* if its rules are so constructed as to make the dependency relation between the nodes of the system a *partial order* and upon execution of these rules the state of the system eventually satisfies a global state predicate. Intuitively, the dependency relation is antisymmetric so that thrashing cannot occur.

A phase is defined to be *closed* if no rules in it are enabled once the state of the system satisfies a global state predicate.

A phase is said to be *stabilizing* if it is convergent and closed [2].

The *write set* of a phase is the set of register fields that are updated in the phase. If the write sets of the phases constituting the protocol are mutually disjoint and each of the phases is individually stabilizing, then the protocol is stabilizing.

We assume that an underlying spanning tree protocol as in [3] or [7] maintains
the consistency of the field parent in the registers. As in [13, 18], our protocols make no assumptions about a fair scheduler and will also work with a distributed scheduler [5, 6]. Although a read/write atomic model is not explicitly assumed in the model, the protocols will also work correctly in such models as in [3, 10, 13]. Refer to NOTRHS in 2.2.1 for an explanation.

The protocols for diameter, centroid and median computation work in two phases each. In the up phase, the value of the metric is computed in each node's up variable using the up variables of its children, so that the up of the root stabilizes to the correct value of the metric. The root then copies its up variable to its down variable. In the down phase, each node copies the down variable of its parent into its down variable, so that down contains the correct value for the metric.

2.2.1 Functions Used in the Protocols

In order to simplify the presentation of the rules in sections 2.2.2 and 2.2.3, we developed the following functions. The function NOTRHS is used to avoid repetition in the rules and make them look cleaner.

For example, Rule R0 would read

\[ r_i.h \neq MAX.CHILD HT(i) + 1 \rightarrow r_i.h := MAX.CHILD HT(i) + 1; \]

This check is done so that the protocol will work correctly even with an unfair scheduler. Rules whose guards are false will not be scheduled and are 'blocked out' of execution. Also, assuming a read/write atomic model, if the value of the variable being read changes in between the read and write atomic steps, the 'correct' value will be written the next time the same rule is executed. There will be a next time because the guard of the rule in question will evaluate to true as a result of the NOTRHS function.

\[ \text{CHILD}(i) \quad (* \text{Returns the set of registers of the children of } i. \text{ Returns the null set if} \ i \text{ is a leaf.} \ \{r_j\} \text{ is a local variable. *}) \]

\[
\begin{align*}
\{r_j\} & := \emptyset; \\
\text{for each } j & \mid (r_j.parent = i)
\end{align*}
\]
add1 \( \{r_j\}, r_j\);  
return \( \{r_j\} \);

\textit{MAX\_CHILD\_HT} \( (i) \) (* Returns the maximum value of the ht of the children of \( i \). Returns 0 if \( i \) is a leaf. *)

\{  
return \( \text{MAX}(\text{CHILD}(i).ht) \);
\}

\textit{MAX2\_CHILD\_HT} \( (i) \) (* Returns the second maximum value of the ht of the children of \( i \). Returns 0 if \( i \) is a leaf. *)

\{  
return \( \text{MAX2}(\text{CHILD}(i).ht) \);
\}

\textit{MAX\_CHILD\_DT} \( (i) \) (* Returns the maximum value of the dt.up of the children of \( i \). Returns 0 if \( i \) is a leaf. *)

\{  
return \( \text{MAX}(\text{CHILD}(i).dt.up) \);
\}

\textit{CENTROID} \((i)\)\(^2\)

\{  
if \( (r_i.ht = [(r_i.dt.down/2)] + 1) \)
then return TRUE;
else return FALSE;
\}

\textit{ROOT} \( (i) \)

\{  
if \( (r_i.parent = 0) \)
then return TRUE;
else return FALSE;
\}

\textit{PARENT} \( (i) \) (* Returns the register of the parent of \( i \) *)

\{  
return \( r_j \mid r_i.parent = j \);
\}

\textit{MEDIAN} \( (i) \)

\(^1\)The add function adds an element to a set if it is not already a member of the set.

\(^2\)For the other centroid, replace [ ] with \( ] \)
\{ 
  if (2 * MAX(CHILD(i).count) > r1.nodes) 
    then return TRUE;
  else return FALSE;
\}

NOTRHS
\{ 
  In the right hand side of the rule, i.e. the ‘assignment statement’, replace := with \neq;
  If (the condition thus got) is TRUE
    then return TRUE;
  else return FALSE;
\}

2.2.2 Diameter and Centroid Protocols

The protocols consist of eight rules, R0 ... R7; R0 ... R3 being for diameter calculation and R4 ... R7 being for centroid identification. The function MAX in R1 and R3 and the function MAX2 in R3 calculate the greatest and second greatest values of their parameters, respectively. These functions return zero when applied to the null set and the singleton set, respectively.

Definition 1 The height of a non-leaf node is one plus the maximum height of its children; the height of a leaf being one.

Definition 2 The diameter of a tree is the number of edges in a longest simple path in the tree.

The diameter protocol ensures that the register field r1.dt.down in each node stabilizes to the value of the diameter of the tree. This occurs in three phases. In Phase 1, rule R0 calculates the height of the node in r1.ht. This rule is straightforward, the ht of a node is one greater than the maximum ht of all its children; the ht of a leaf being 1.

Rule R1 performs a convergecast so that the variable dt.up at the root stabilizes to the value of the diameter of the tree. This is Phase 2. dt.up at each node is the
sum of the two greatest $ht$ values of its children or the greatest $dt.up$ value of its
children which is the diameter of the subtree rooted at the node. $dt.up$ of a leaf is
zero.

Rules $R2$ and $R3$ constituting Phase 3 broadcast the diameter, so that the value
of $dt.down$ at each node equals the diameter of the tree. Each node copies $dt.down$
from $dt.down$ of its parent ($R3$), the root copying it from its own $dt.up$ instead ($R2$).

**Definition 3** A node in a tree is called a centroid if it is a middle node in a longest
simple path in the tree.

The centroid protocol has two phases. In Phase 1, a convergecast of the index of
the centroid occurs ($R4$ and $R5$). One of the two centroids of the tree (or the only
one: refer to Lemma 2.6) is the node whose $ht$ equals $\lceil \frac{dt.down}{2} \rceil + 1$ ($R4$).

In Phase 2 ($R6$ and $R7$), the index of the centroid is broadcast to all nodes.
Each node copies $center.down$ from $center.down$ of its parent ($R7$); the root copies
$center.down$ from its own $center.up$ ($R6$).

\[
\begin{align*}
\{\text{Compute } ht \text{ values} \} \\
R0 :: & \ NOTRHS \rightarrow r_i.ht := MAX.CHILD.HT(i) + 1
\end{align*}
\]

\[
\begin{align*}
\{\text{Convergecast the diameter} \}
&& R1 :: & NOTRHS \rightarrow r_i.dt.up := MAX(MAX.CHILD.HT(i) + MAX2.CHILD.HT(i), MAX.CHILD.DT(i))
\end{align*}
\]

\[
\begin{align*}
\{\text{Broadcast the diameter} \}
&& R2 :: & ROOT(i) \land NOTRHS \rightarrow r_i.dt.down := r_i.dt.up
\end{align*}
\]

\[
\begin{align*}
&& R3 :: & \sim ROOT(i) \land NOTRHS \rightarrow r_i.dt.down := PARENT(i).dt.down
\end{align*}
\]

\[
\begin{align*}
\{\text{Convergecast the centroid } \}
&& R4 :: & CENTROID(i) \land NOTRHS \rightarrow r_i.center.up := i
\end{align*}
\]

\[
\begin{align*}
&& R5 :: & \sim CENTROID(i) \land NOTRHS \rightarrow r_i.center.up := MAX(CHILD(i).center.up)
\end{align*}
\]

\[
\begin{align*}
\{\text{Broadcast the centroid} \}
&& R6 :: & ROOT(i) \land NOTRHS \rightarrow r_i.center.down := r_i.center.up
\end{align*}
\]
\[11
\]
\[
\begin{align*}
|| R7 :: & ~ \text{ROOT}(i) \land \text{NOTRHS} \\
\rightarrow & \rightarrow r_i.\text{center.down} := \text{PARENT}(i).\text{center.down}
\end{align*}
\]

2.2.3 Median Protocol

The protocol consists of seven rules, \( R8 \ldots R14 \). The function \( \text{MAX} \) in \( R11 \) and \( R12 \) calculates the greatest value of its parameters, and the function \( \text{SUM} \) in \( R8 \) calculates the sum of its parameters. Both these functions return 0 when applied to the null set.

The protocol ensures that the register field \( r_i.\text{median.down} \) in each node stabilizes to the index of one of the medians of the tree. This occurs in four phases. In Phase I, rule \( R8 \) calculates the \textit{count} at each node \( i \), which is the number of nodes in the subtree rooted at \( i \). At the end of Phase I, the value of \textit{count} at the root is the count of nodes in the tree. In Phase II, the value of \textit{nodes} at each node \( i \) stabilizes to the value of the number of nodes in the tree. The rules for Phase II involve the root copying its \textit{nodes} from its \textit{count} (\( R9 \)) and each node copying \textit{nodes} from the variable \textit{nodes} of its parent (\( R10 \)). In Phase III, the median is computed using the rules \( R11 \) and \( R12 \).

\textbf{Definition 4} A node in a tree is called a median if the sum of the distances from this node to all other nodes in the tree is the least possible.

These rules perform a convergecast so that the value of \( \text{median.up} \) at the root stabilizes to the node index of one of the medians of the tree. A node \( i \) checks if twice the greatest \( r_j.\text{count} \) of all its children is less than \textit{nodes}, and if so it declares itself the median by setting \( r_i.\text{median.up} \) to its own index (\( R11 \)). Otherwise, it copies the greatest \( \text{median.up} \) from its children into \( r_i.\text{median.up} \) (\( R12 \)). The value of \( \text{median.up} \) at the root stabilizes to the index of the median of the tree.

In Phase IV, a broadcast of the index of the median is done. The root copies its \( \text{median.up} \) variable into its \( \text{median.down} \) variable (\( R13 \)). Each non-root node copies
median.down from its parent's median.down (R14). Thus the value of median.down at each node stabilizes to the index of the median of the tree.

{Compute count values}
R8 :: NOTRHS → ri.count := SUM(CHILD(i).count) + 1

{Broadcast value of nodes}
|| R9 :: ROOT(i) ∧ NOTRHS → ri.nodes := ri.count
|| R10 :: ~ ROOT(i) ∧ NOTRHS → ri.nodes := PARENT(i).nodes

{Convergecast the median}
|| R11 :: MEDIAN(i) ∧ NOTRHS → ri.median.up := i
|| R12 :: ~ MEDIAN(i) ∧ NOTRHS
→→ ri.median.up := MAX(CHILD(i).median.up)

{Broadcast the median}
|| R13 :: ROOT(i) ∧ NOTRHS → ri.median.down := ri.median.up
|| R14 :: ~ ROOT(i) ∧ NOTRHS
→→ ri.median.down := PARENT(i).median.down

2.3 Proof of Correctness

To prove that a protocol is correct, we prove that each phase constituting the protocol is convergent and closed. Closure is proved by defining a global state predicate for each phase and proving that once this state is reached, no rule in the phase is enabled for any node. In each phase, we prove convergence by induction. This is acceptable since every phase is either up convergent or down convergent. An up convergent phase maintains a linear order \(<\) between the nodes of the system such that
\[(\forall i)(\forall j) (i < j) \iff r_i.ht < r_j.ht\]
For a down convergent phase, the order \(<\) is such that
\[(\forall i)(\forall j) (i < j) \iff r_i.ht > r_j.ht\]
Intuitively, information flow is upwards towards the root for an up convergent phase, while it is towards the leaves for a down convergent phase. For an up convergent
phase, the leaves are the minimal elements of the partial order while for a down convergent phase, the root is the minimal element. Hence, for an up convergent phase, induction is done with the leaves as the bases, while for a down convergent phase, the root forms the basis of the induction.

Convergence is guaranteed even with an unfair scheduler because the nodes form a partial order and thus the scheduler is constrained to schedule those nodes which have not stabilized yet. Therefore, convergence will occur in finite time.

Distributed scheduling permits simultaneous actions by different nodes. Our protocols work with such a scheduler because the dependency graph of the nodes is acyclic. Thus one node executing actions concurrently with another cannot interfere with, and undo the actions of, another.

2.3.1 Diameter and Centroid Protocols

DCSec The following global state predicates are defined for the phases in these protocols:

\[ G_h : \forall i, r_i. ht = \text{MAX}_CHILD_HT(i) + 1 \]
\[ G_{d1} : G_h \land (\forall i, r_i. dt.up = \text{MAX}(\text{MAX}_CHILD_HT(i) + \text{MAX}_2\text{CHILD_HT}(i), \text{MAX}_CHILD_DT(i))) \]
\[ G_{d2} : G_{d1} \land (\forall i, (\text{ROOT}(i) \land (r_i. dt.down = r_i. dt.up)) \lor (\sim \text{ROOT}(i) \land (r_i. dt.down = \text{PARENT}(i). dt.down))) \]
\[ G_{c1} : G_{d2} \land (\forall i, (\text{CENTROID}(i) \land (r_i. center. up = i)) \lor (\sim \text{CENTROID}(i) \land (r_i. center. up = \text{MAX}(\text{CHILD}(i). center. up)))) \]
\[ G_{c2} : G_{c1} \land (\forall i, (\text{ROOT}(i) \land (r_i. center. down = r_i. center. up)) \lor (\sim \text{ROOT}(i) \land (r_i. center. down = \text{PARENT}(i). center. down))) \]

Lemma 2.1 The phase \{Compute ht values\} is stabilizing.

Proof: It is evident that the only rule for this phase, R0, is not enabled in the state \( G_h \), so the phase is closed. This phase is up convergent by inspection, this may be proved inductively using the definition of \( h \) of a node.

Lemma 2.2 The value of \( r_i. dt. up \) in each node \( i \) stabilizes to the diameter of the subtree rooted at \( i \) after a finite number of applications of Rule R1.
Proof: \{Convergecast dt values\} is up convergent since the guard of \( R1 \) for \( i \) is an expression over registers \( r_j \) of the children \( j \) of \( i \). The guard of \( R1 \) is not true in state \( G_{d1} \), so this phase is closed.

A formal proof of convergence by induction on the height of the subtree rooted at \( i \) follows.

**Basis:** The minimal elements are the leaves. If \( i \) is a leaf, rule \( R1 \) stores in \( r_i.dt.up \) the value zero which is the diameter of the tree rooted at \( i \). Thus the basis case is true.

**Induction Hypothesis:** Assume that \( R1 \) converges \( r_j.dt.up \) to the diameter of the subtree rooted at \( j \) where \( js \) are those nodes which have height \( h \geq 1 \).

**Induction Step:** We now establish that \( R1 \) converges \( r_i.dt.up \) to the diameter of the subtree rooted in \( i \) when the subtree has height \( h + 1 > 1 \).

Let \( \rho \) be a largest simple path in the subtree rooted at \( i \). Since the subtree rooted at \( i \) has height \( > 1 \), it must have at least one child. We deal with two cases: in one \( i \) has exactly one child and in the other it has more than one child.

**Case 1:** Node \( i \) has exactly one child (node \( j \)).

In this case, either path \( \rho \) has \( i \) as an endpoint, or it does not include \( i \). If \( i \) is an endpoint of \( \rho \), the diameter of the tree rooted at \( i \) is \( r_i.ht \) which by definition is greater than or equal to the diameter of the subtree rooted at \( j \). By the induction hypothesis, \( r_j.dt.up \) has converged, so that \( r_i.dt.up \) also converges.

If \( \rho \) does not include \( i \), the diameter of the subtree rooted at \( i \) equals the diameter of the subtree rooted at \( j \) which by the hypothesis, has already converged. Since path \( \rho \) does not include node \( i \), \( r_j.ht \) must be less than or equal to the diameter of the subtree rooted at \( j \). Thus, in either case, the variable \( r_i.dt.up \) converges to the diameter of the subtree rooted at \( i \).

**Case 2:** Node \( i \) has more than one child.

Again, either path \( \rho \) goes through node \( i \) or it does not include \( i \). In the former case, the rule \( R1 \) computes \( r_i.dt.up \) as the sum of the largest two heights of the children of \( i \) (the value of \( ht \) at all nodes has stabilized), which by definition is greater than or equal to the diameter of any subtree rooted at a child of \( i \). By the induction
hypothesis, the value of $r_j.dt.up$ has converged to the value of the diameter of the subtree rooted at $j$ for every child $j$ of $i$.

In the latter case, the diameter of the subtree rooted at $i$ is equal to the diameter of the subtree of a child $j$ of $i$. By definition, this value is greater or equal to the sum of the largest two heights of the children of $i$ (which have already stabilized). In either case, it is simple to verify that the value of $r_i.dt.up$ converges to the value of the diameter of the subtree rooted at $i$. $\square$

**Corollary 1** The variable $dt.up$ at the root stabilizes to the value of the diameter of the tree after a finite number of applications of the rules $R0$ and $R1$.

*Proof:* Follows directly from Lemma 2.2. $\square$

**Lemma 2.3** The variable $dt.down$ in each node $i$ stabilizes to the value of the diameter of the tree after a finite number of applications of $R2$ and $R3$.

*Proof:* The phase is closed with respect to $G_{d2}$ since rules $R2$ and $R3$ are not enabled when the system is in this state.

{Broadcast $dt$ values} is down convergent since since the guards of $R2$ and $R3$ are expressions over registers $r_j$ of the parent of $i$, if one exists. Proof by induction follows:

*Basis:* The root is the basis of the induction. By Corollary 1, the value of $r_1.dt.down$ eventually becomes equal to the diameter of the tree. By applying $R2$, the root sets register field $dt.down$ equal to $dt.up$. Hence the value of $r_1.dt.down$ equals the diameter of the tree.

*Induction Hypothesis:* Assume that all nodes at level $l$ have $dt.down$ equal to the diameter of the tree.

*Induction Step:* We now establish that all nodes at level $l + 1$ will eventually have $dt.down$ equal to the diameter of the tree. The down convergence of this phase implies that the nodes at level $l + 1$ depend only on those at levels $l$ and below, so that if those at level $l$ have converged, then so do those at level $l + 1$. $\square$

**Theorem 2.1** The diameter protocol is correct.
Proof: The write sets of the phases of this protocol are \{r_i,ht\}, \{r_i,dt.up\} and \{r_i,dt.down\}. These are mutually disjoint, by observation.

Hence, the diameter protocol is correct since its individual phases have been proven correct by Lemma 2.1, Corollary 1, and Lemma 2.3.

\textbf{Lemma 2.4} [Korach, Rotem, and Santoro [15]] The statement \(r_i,ht = \left\lfloor \frac{r_i,dt.down}{2} \right\rfloor + 1\) holds for only one node of the tree \(T\), that node being a centroid of the tree.

Proof: It is simple to see that for at least one centroid \(P_c\) of the tree, the statement \(r_c,ht = \left\lfloor \frac{r_c,dt.down}{2} \right\rfloor + 1\) holds.

We will prove Lemma 2.4 by contradiction, by assuming that there is another node \(P_{c1}\) in the tree for which the statement \(r_{c1},ht = \left\lfloor \frac{r_{c1},dt.down}{2} \right\rfloor + 1\) holds.

Since the nodes \(P_c\) and \(P_{c1}\) are not identical, but have the same height, it cannot be that one is a predecessor of the other in the tree. Let \(P_x\) be the node which is the closest ancestor of both \(P_c\) and \(P_{c1}\). Then the path consisting of a longest path from \(P_c\) to a leaf, plus the path from \(P_x\) to \(P_c\), plus the path from \(P_x\) to \(P_{c1}\), plus a longest path form \(P_{c1}\) to a leaf, is a simple path and has length greater than or equal to \(2 \times \left\lfloor \frac{r_x,dt.down}{2} \right\rfloor + 2 > r_x,dt.down\). Since this contradicts the definition of diameter of a tree, it cannot be that the statement \(r_i,ht = \left\lfloor \frac{r_i,dt.down}{2} \right\rfloor + 1\) holds for more than one node in \(T\).

\textbf{Lemma 2.5} The statement \(r_i,ht = \left\lfloor \frac{r_i,dt.down}{2} \right\rfloor + 1\) holds for only one node of the tree, that node being a centroid of the tree.

Proof: The proof is similar to the proof of Lemma 2.4.

\textbf{Lemma 2.6} [Deo [8]] There are at most two centroids in a tree.

Proof: Refer to [8] for the proof.

\textbf{Lemma 2.7} There may be more than one longest path in a tree, but all of them contain the centroid(s) in the tree.

Proof: Refer to [8] for the proof.
Lemma 2.8  The value of \( r_i.\text{center.up} \) at the root stabilizes to the index of one of the centroids of the tree after a finite number of applications of rules R4 and R5.

Proof: It is evident that the guards of R4 and R5 are not enabled once the system reaches \( G_{c1} \). Hence this phase is closed with respect to \( G_{c1} \).

A proof of convergence follows:
For at least one node \( P_c \), the expression \( (\hat{r}_c.\text{ht} = \lceil \frac{r_c.\text{dist} \cdot \text{down}}{2} \rceil + 1) \) is true (Lemma 2.4).

This expression forms part of the guards of R4 and R5 and hence will be true for at least one node, namely, one of the centroids of the tree. This node sets its register field \( r_c.\text{center.up} \) to its index.

Since this phase is up convergent, it may be proved by induction using this node as the basis that the root eventually gets the centroid's index in its register field \( \text{center.up} \).

Lemma 2.9  The variable center.down in each node i stabilizes to the index of the centroid of the tree after a finite number of applications of R6 and R7.

Proof: It is evident that the guards of R6 and R7 are not enabled once the system reaches \( G_{c2} \). Hence this phase is closed with respect to \( G_{c2} \).

The phase \{Broadcast the centroid\} being down convergent, an inductive proof may be constructed for this lemma along the lines of Lemma 2.3.

Theorem 2.2  The centroid protocol is correct.

Proof: Notice that the centroid protocol includes the three phases of the diameter protocol, apart from the two phases that find the centroid. By inspection, the write sets of the five phases are mutually disjoint. Thus, the centroid protocol is correct since its individual phases have been proved correct by Lemma 2.1, Corollary 1, lemmas 2.3, 2.8 and 2.9.

2.3.2  Median Protocol

The following global states are defined for the phases in this protocol and will be used in the lemmas that follow:
Lemma 2.10 \{Compute count values\} stabilizes the value of $r_i.count$ at each node $i$ to the count of the nodes in the subtree rooted at $i$.

**Proof:** It is easy to verify that this phase is closed when the system reaches the state $G_c$. This phase is up convergent and a proof for this is inductive with the leaves as the bases. □

Lemma 2.11 \{Broadcast value of nodes\} stabilizes the value of $r_i.nodes$ in each node to the count of nodes in the tree.

**Proof:** We may construct a proof of this lemma along the lines of Lemma 2.3. The system is closed when it reaches the state $G_n$. □

Lemma 2.12 A median of the tree, $P_m$, satisfies the condition, $2ct(j) > n$ where $ct(j)$ is the maximum count of the children of $P_m$ as defined below and $n$ is the number of nodes in the tree.

**Proof:** Before we can prove this lemma, we will need the following definitions and observations:

**Definition 5** The count of a leaf is 1, and the count of a non-leaf node is one plus the sum of the counts of its children.

**Definition 6** The total distance from node $i$ to all the nodes in a tree is the sum of the lengths of the path from node $i$ to each node in the tree.
Observation 1 [Korach, Rotem and Santoro [15]] If node $i$ in a tree has total distance to all nodes in $T$ equal to $\text{dis}(i)$, then the corresponding value for its child $j$ is

$$\text{dis}(j) = \text{dis}(i) + n - 2\text{ct}(j),$$

where $\text{ct}(j)$ is the count of $j$.

This formula follows from the fact that the length of the path from a node $i$ to a node $k$ which is not $j$ or a descendant of $j$ is one less than the length of the path from node $j$ to node $k$; the length of the path from $i$ to a node $k$ which either is node $j$ or a descendant of $j$ is one more than the length of the path from $j$ to node $k$; and the number of nodes in the subtree rooted at $i$ is $\text{ct}(i)$.

Observation 2 If node $i$ in the tree $T$ has $2\text{ct}(i) > n$ then for at most one of its children $j$, $2\text{ct}(j) \geq n$. When such a child exists, $\text{dis}(j) \leq \text{dis}(i)$.

This follows from the fact that $n$ is the total number of nodes in the tree and $\text{ct}(i) \leq n$.

The proof of Lemma 2.12 can now be stated. Assuming we know that all medians of the tree are in the subtree rooted at $i$ and that $2\text{ct}(i) > n$. Then by Observation 2, one of the following three cases applies:

Case 1: There is one child $j$ of $i$ for which $2\text{ct}(j) > n$. Then by Observation 1, $\text{dis}(i) < \text{dis}(j)$, and $\text{dis}(k) > \text{dis}(i)$ for all the other children $k$ of $i$. Therefore, all medians are in the subtree rooted at $j$ and $2\text{ct}(j) > n$.

Case 2: There is only one child $j$ of $i$ for which $2\text{ct}(j) = n$. By Observation 1, $\text{dis}(i) = \text{dis}(j)$ and $\text{dis}(k) > \text{dis}(i)$ for all other children $k$ of $i$. Therefore $j$ and $i$ are the medians of the tree.

Case 3: There is no child $j$ for which $2\text{ct}(j) \geq n$. By Observation 1, there cannot be a median in the subtrees rooted at any child of $i$. Therefore node $i$ is the median of the tree. □

Lemma 2.13 [Korach, Rotem, and Santoro [15]] There are at most two medians in a tree.
Proof: Refer to [15] for the proof.

Lemma 2.14 \{Convergecast the median\} stabilizes the value of \( r_i\_median\_up \) in the root to the index of the median.

Proof: According to Lemma 2.12, for at least one node \( P_m \) in the tree, the statement 
\[
(\forall j) \ (r_j\_parent = m) \ (2 \cdot \max(r_j\_count) < r_m\_nodes)
\]
will be true, this node being one of the medians of the tree. This expression being part of the guards of \( R11 \) and \( R12 \), the node \( P_m \) sets \( r_m\_median\_up \) to its node index. Using this node as the basis, we may prove up convergence of this phase. The proof would be similar to that of Lemma 2.8. The system is closed when it reaches the state \( G_{m1} \).

Lemma 2.15 \{Broadcast the median\} stabilizes the value of \( r_i\_median\_down \) in each node to the index of one of the medians.

Proof: The proof for this lemma is again identical to that of Lemma 2.3. When the system reaches state \( G_{m2} \), it is closed since no rules are enabled.

Theorem 2.3 The median protocol is correct.

Proof: Notice that the write sets of the phases constituting the median protocol are disjoint. Since we have proved that each phase is individually stabilizing, the median protocol is correct.

2.3.3 Complexity

Lemma 2.16 The time complexity of any phase is proportional to the length of the longest dependency chain of the partial order for the phase.

Proof: The minimal elements of the partial order stabilize immediately. Each non-minimal element depends directly or indirectly on those elements that precede it in the partial order. Thus the time taken for a phase to stabilize increases with increasing length of the longest dependency chain.
2.4 Conclusions

The protocols presented in this chapter are self-stabilized algorithms for calculating the diameter and locating the centroids and medians of a distributed tree structured network. They provide fault-tolerant means of drawing topological information about a tree network. No assumptions are made about the fairness of the scheduler. A distributed scheduling model may also be assumed for the network and the protocol will still work correctly. The model assumed has very weak atomicity. The ideas behind these algorithms could conceivably be extended to finding the diameter, centroids, and medians of a general graph network; this would be a challenging problem.
Chapter 3

MUTUAL EXCLUSION

3.1 Introduction

Dijkstra [9] pioneered the study of Self-Stabilization in distributed systems in 1974 when he studied mutual exclusion among finite state machines connected in a ring. He defined the privilege of a machine as the ability to change its current state. This ability is based on a boolean predicate involving its current state and those of its neighbors. Only when the machine has a privilege can it change its current state: this action is referred to as a move.

In order for the system to be self-stabilizing, the legal states must satisfy the following conditions:

[P1] There must be at least one privilege in the system (no deadlock).
[P2] Every move from a legal state must again put the system into a legal state (closure).
[P3] During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
[P4] Given any two legal states, there is a series of moves that change one legal state to the other (reachability).
Dijkstra considered a legal state as one in which exactly one machine enjoys a privilege. This corresponds to a form of mutual exclusion, because the privileged process is the only process that is allowed in its critical section. Once the process leaves the critical section, it passes the privilege to one of its neighbors.

A great deal of work has been done in the area of self-stabilizing mutual exclusion. Mutual exclusion can be achieved using privileges [9] or tokens [5]. Most of the work on self-stabilizing mutual exclusion assumes that the network has a certain topology. In [10], a self-stabilizing algorithm on the spanning tree of a distributed system was presented under the Read/Write demon. Because the spanning tree of a distributed system can also be obtained by some self-stabilizing algorithms [3, 7], the mutual exclusion of the spanning tree has no restriction on the topology of the distributed system. It can be applied to any distributed system. However, unbounded number of variables were used for each processor in the algorithm. In this chapter, we present a bounded variable self-stabilizing algorithm for mutual exclusion on a tree structured distributed system. The new self-stabilizing spanning tree algorithm presented in this chapter is based on one of Dijkstra's algorithms [9] and its variation by Ghosh [12].

The rest of the chapter is as follows. The next section presents the system model and the algorithm. Section 3.3 provides the proof of correctness while Section 3.4 gives the conclusion.

3.2 Description of the Protocol

3.2.1 System Model

A distributed system may be conceived of as a graph: the processors constitute the nodes of the graph and the links between them are the edges of the graph. Existing methods can be used to construct a spanning tree of the graph in a self-stabilizing manner. Our protocol then provides for Mutual Exclusion on the tree via a system of privileges. A node is considered to have a privilege when its rules enable it to
change its state. When the privilege occurs because of the parent, (Rules R2 and R3) it is considered a \textit{P\_PRIV} privilege. When the privilege occurs because of the \texttt{CurrentChild} (Rules R1 and R5), it is a \textit{C\_PRIV} privilege.

The read/write register \( r_i \) of node \( P_i \) contains the following fields:

- \( r_i.id \) has the id of \( P_i \)
- \( r_i.parent \) has the node index of the parent of \( P_i \) except for the root, which has zero
- \( r_i.state \) has the state of \( P_i \)
- \( r_i.FirstChild \) has a pointer to the first child of \( P_i \)
- \( r_i.CurrentChild \) has a pointer to the child of \( P_i \) which we are considering
- \( r_i.Sibling \) has a pointer to the sibling of \( P_i \)

The state of \( P_i \) is in \( \{0,2\} \) if \( P_i \) is the root of the tree, \( \{1,3\} \) if \( P_i \) is a leaf, and in \( \{0,1,2,3\} \) otherwise.

### 3.2.2 The Algorithm

In order to simplify the presentation of the rules of the algorithm the following macros are used in the algorithm:

\texttt{NEXTCHILD}(i)

```c
{
    return r_i.CurrentChild;
}
```

\texttt{LASTCHILD}(i)

```c
{
    if \((\text{NEXTCHILD}(i) = r_i.FirstChild)\) 
    then return \texttt{TRUE};
    else return \texttt{FALSE};
}
```

\texttt{ISCURRENTCHILD}(i)

```c
{
    if \((r_i.parent.CurrentChild.id = r_i.id)\)
    then return \texttt{TRUE};
    else return \texttt{FALSE};
}
```
The protocol consists of six rules, \( R_0 \ldots R_5 \); \( R_0 \) and \( R_1 \) being for the root node, \( R_2 \) for the leaf nodes and \( R_3 \ldots R_5 \) being for interior nodes.

\[
\begin{align*}
\text{(For the root node)} & \quad R_0 : (r_i.\text{CurrentChild.state} = r_i.\text{state} + 1) \land \sim \text{LASTCHILD}(i) \\
& \quad \quad \rightarrow \text{NEXTCHILD}(i) \\
\| \quad R_1 : (r_i.\text{CurrentChild.state} = r_i.\text{state} + 1) \land \text{LASTCHILD}(i) \\
& \quad \quad \rightarrow \text{NEXTCHILD}(i); \ r_i.\text{state} := r_i.\text{state} + 2
\end{align*}
\]

\[
\begin{align*}
\text{(For the leaf nodes)} & \quad R_2 : (r_i.\text{parent.state} = r_i.\text{state} + 1) \land \text{ISCURRENTCHILD}(i) \\
& \quad \quad \rightarrow r_i.\text{state} := r_i.\text{state} + 2
\end{align*}
\]

\[
\begin{align*}
\text{(For the interior nodes)} & \quad R_3 : (r_i.\text{parent.state} = r_i.\text{state} + 1) \land \text{ISCURRENTCHILD}(i) \\
& \quad \quad \rightarrow r_i.\text{state} := r_i.\text{state} + 1 \\
\| \quad R_4 : (r_i.\text{CurrentChild.state} = r_i.\text{state} + 1) \land \sim \text{LASTCHILD}(i) \\
& \quad \quad \rightarrow \text{NEXTCHILD}(i) \\
\| \quad R_5 : (r_i.\text{CurrentChild.state} = r_i.\text{state} + 1) \land \text{LASTCHILD}(i) \\
& \quad \quad \rightarrow \text{NEXTCHILD}(i); \ r_i.\text{state} := r_i.\text{state} + 1
\end{align*}
\]

Initially, the processors in the system may have any possible state values, hence there may exist more than one privilege. The goal of the algorithm is to reduce the number of privileges to one in a finite number of steps.

### 3.3 Proof of Correctness

**Lemma 3.1** There is at least one privilege in the system.

**Proof:** This may be proved by contradiction. Assume that no privilege exists in the system. Since the root is constrained to have a state of 0 or 2, its \( \text{CurrentChild} \) has a state of 0 or 2 because it does not have a privilege. This argument can be extended to all the \( \text{CurrentChildren} \). But the leaf

\footnote{All + operations are mod 4}
is constrained to have a state of 1 or 3. Thus there is at least one privilege in the system and there is no deadlock.

We will now prove that every process has to make a move in a finite amount of time.

**Lemma 3.2** If a parent process does not change its state $s$ and one of its children, $P_c$ gets $P\_PRIV$ privileges twice, then at least one privilege is lost in the subtree rooted at $P_c$.

**Proof:** By rule $R3$, when $P_c$ gets a $P\_PRIV$ privilege, its sets its state equal to $s$, the state of its parent. If $P_c$ can get a $P\_PRIV$ privilege again, then it must have changed its state in the following sequence of moves because of $C\_PRIV$ privileges:

$$s \rightarrow s + 1 \rightarrow s + 2 \rightarrow s + 3$$

At least one privilege is lost for the subtree rooted at $P_c$ when changing from $s + 1$ to $s + 2$. 

**Corollary 2** If a process $P_c$ does not move and its parent $P_p$ gets $C\_PRIV$ privilege due to $P_c$ twice, then at least one privilege is lost in the path from the root to $P_p$.

**Proof:** This follows by an argument similar to the proof of Lemma 3.2.

**Lemma 3.3** Every process must make a move in a finite amount of time.

**Proof:** Proof by contradiction.

Suppose process $P_i$ does not move. Then, its children can move only a finite number of times. By Lemma 3.2, for each round of changing states of the child, at least one privilege will be lost in the subtree and the total number of privileges in the subtree will not increase. After the children of this process stop, the next generation of descendants will also stop after a finite number of moves. Finally, all the processes in the subtree rooted at $P_i$ cannot move. Similarly, by Corollary 2, all the processes from the root to $P_i$ will not be able to move, thus forcing $P_i$ to make a move.

We will now prove that mutual exclusion is guaranteed by the protocol. This will be done by using an $l\_value$, for each node. A variable $l\_value$ is not needed in the
algorithm, but is just used to show the self-stabilization of the algorithm. Initially, the \textit{l.value} of a process will be its \textit{level} in the spanning tree, going by its standard definition. When a process \( P_i \) changes its state due to a \textit{C_PRIV} privilege, its \textit{l.value} is updated to the minimum of the \textit{l.values} of its children. When \( P_i \) changes its state due to a \textit{P_PRIV} privilege, it changes its \textit{l.value} to that of its parent.

\textbf{Lemma 3.4} The \textit{l.values} in the system are nondecreasing from the root to the leaves in the tree.

\textit{Proof:} Originally, the \textit{l.values} are increasing from the root to the leaves. When a process changes its \textit{l.value}, it either selects the smallest of its children's \textit{l.values} or its parent's \textit{l.value}. In either case, the relation is preserved. \( \square \)

\textbf{Lemma 3.5} Eventually, all \textit{l.values} will be the same.

\textit{Proof:} We will prove this by inducting on the number of nodes in the network.

\textbf{Basis:} The network only has two nodes.

Initially, the \textit{l.values} are 1 and 2. By Lemma 3.1, there must be a privilege in the system, so one of these processes must make a move. When one of the processes makes a move, its \textit{l.value} will be changed. At this point, all \textit{l.values} will be the same.

\textbf{Induction Hypothesis:} Assume that for a network of \( k \) nodes, all \textit{l.values} will be equal eventually.

Now, we show that all \textit{l.values} will be equal eventually in a network of \( k + 1 \) nodes. Without loss of generality, assume that there are at least three levels in the tree. We will prove the lemma for a tree having less than three levels in a lemma that follows. Consider the last level of the tree. The nodes on this level are leaf nodes. Choose any leaf, and consider the group of nodes made up of this leaf, its parent, and its siblings (the siblings are leaves as well). The only way in which the parent in the group can change its level is through its last child or by its parent. Eventually, the parent in the group will move because of its parent. It can only make a finite number of moves because of its children (they are all leaves), and after these moves,
it will eventually make another move (Lemma 3.3). This move must be because of its parent. When this occurs, this group of nodes can be considered as one node because the parent in the group will never change its \textit{L.value} because of its children. After the move (due to its parent), it has \textit{l.value} \(l\). In order for it to change its \textit{l.value} because of its children to \(m\), it must change its state. So, Step 6 must be executed. This implies that all other children have moved. Since \(l\) can not decrease and the other children have \textit{l.values} of \(l\), the parent's \textit{l.value} will stay the same. So, the rest of this network sees this group of nodes as one node because its \textit{l.values} come from the rest of the tree. This network now has less than \(k+1\) nodes. Eventually, all nodes will have the same \textit{l.value}. 

\begin{observation}
Lemma 3.5 trivially holds if there are only two levels (no interior nodes).
\end{observation}

\begin{lemma}
When all \textit{l.values} in the system are the same, there will only be one privilege in the system.
\end{lemma}

\begin{proof}
Proof by contradiction.
\end{proof}

\begin{theorem}
The algorithm is a self-stabilizing mutual exclusion algorithm.
\end{theorem}

\begin{proof}
This follows directly from Lemma 3.6, and the cycle of legal states.
\end{proof}

\subsection{Conclusions}

In this chapter, we presented an algorithm for self-stabilizing distributed mutual exclusion on a spanning tree of the distributed system. The algorithm can be applied to any connection structure of the distributed system, since the spanning tree of a general graph can be obtained in a self-stabilizing manner using existing algorithms [3, 7]. The algorithm can tolerate both transient errors and node failures. If a node fails, the spanning tree will be automatically recalculated (since the spanning
tree algorithm is self-stabilizing). Once the spanning tree is rebuilt, the system will converge to a state where only one privilege exists. So, the mutual exclusion algorithm is extremely fault tolerant.
A distributed system consists of loosely connected machines which communicate with each other through shared memory and/or message passing in order to achieve a common goal. Reconfiguration, coordination loss or mode change may cause the global system state to become illegal and lose the ability to achieve this common goal. Self-stabilization allows the system to regain coordination between its processors in the event of such a fault. A system is said to be self-stabilizing iff starting from some global state, legal or illegal, the system will converge to a legal state automatically and in a finite number of steps.

Self-stabilization has been applied to a number of areas since its introduction in 1974 by Dijkstra. This thesis applied self-stabilization to several problems in tree-based systems such as the calculation of the diameter of a tree, identification of the centroid and median and also for achieving mutual exclusion.

In Chapter 2, a set of rules to calculate the diameter of a tree and to identify the centroid and median was presented. Such metrics are necessary for most network routing protocols; their self-stabilizing nature makes them tolerant to transient failures and thus applicable to dynamic networks.

In Chapter 3, an algorithm to achieve mutual exclusion on a set of processors forming a tree structured network was presented. This algorithm is simple and can
easily be adapted to run on a general network by pipelining it to an algorithm that will calculate the spanning tree of a general graph. The self-stabilizing version is able to detect any network errors and converge to a legal configuration in a finite amount of time without the need for user intervention.

Self-stabilization is an evolving paradigm in fault-tolerant computing. There are several reasons why self-stabilizing algorithms are better than traditional algorithms:

- The algorithm runs continually (no initiation of the algorithm needs to be done).
- No initialization of the local variables needs to be done, because a self-stabilizing algorithm does not require any initialization.
- The statements in the algorithm can be executed in any order, and the system will still stabilize.
- The algorithm automatically tolerates transient errors (shared memory faults, message corruption)

These reasons, along with the fact that both sets of rules presented are very simple, ensure that they can easily be embedded in real life network algorithms to make them more fault tolerant.
Bibliography


