Compact routing in fault-tolerant distributed systems

James Edward Lawrence

University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/3334

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.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
COMPACT ROUTING IN FAULT-TOLERANT

DISTRIBUTED SYSTEMS

by

James Edward Lawrence

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 1997
The thesis of James Edward Lawrence for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kumar Datta, Ph.D.

Examinining Committee Member, Laxmi P. Gewali, Ph.D.

Examinining Committee Member, Kazem Taghva, Ph.D.

Graduate Faculty Representative, Ashok Iyer, Ph.D.

Dean of the Graduate College, Ronald W. Smith, Ph.D.

University of Nevada, Las Vegas
August 1997
ABSTRACT

A compact routing algorithm is a routing algorithm which reduces the space complexity of all-pairs shortest path routing. Compact routing protocols in distributed systems have been studied extensively as an attractive alternative to the traditional method of all-pairs shortest path routing. The use of compact routing protocols have several advantages. Compact routing schemes are not only more memory-efficient, but provide faster routing table lookup, more efficient broadcast scheme, and allow for a more scalable network. These routing schemes still maintain optimal or near-optimal routing paths. However, most of the compact routing protocols are not fault-tolerant. This thesis will first report the recent developments in the compact routing research. Several new methods for compact routing in fault-tolerant distributed systems will be presented and analyzed. The most important feature of the algorithms presented in this thesis is that they are self-stabilizing. The self-stabilization paradigm has been shown to be the most unified and all-inclusive approach to the design of fault-tolerant systems. Additionally, these algorithms will address and solve several problems left unsolved by previous works. Relabelable and non-relabelable networks will be considered for both specific and arbitrary topologies.
TABLE OF CONTENTS

ABSTRACT ....................................................................................................................... iii
ACKNOWLEDGMENTS ..................................................................................................... vi
CHAPTER 1 INTRODUCTION ........................................................................................ 1
  1.1 Distributed Systems ............................................................................................... 1
  1.2 Fault-Tolerance ..................................................................................................... 2
  1.3 The Routing Problem ......................................................................................... 3
CHAPTER 2 SELF-STABILIZING HIERARCHICAL ROUTING ...................................... 5
  2.1 Previous Work ....................................................................................................... 6
  2.2 Model and Notations ........................................................................................... 6
  2.3 Hierarchical Routing .......................................................................................... 8
  2.4 Fault-Tolerant Hierarchical Routing ................................................................... 9
    2.4.1 Area Partitioning ....................................................................................... 10
    2.4.2 Dynamic Memory Allocation ..................................................................... 11
  2.5 Self-Stabilizing Hierarchical Routing Algorithm ............................................. 13
    2.5.1 Variables and Macros ............................................................................... 14
    2.5.2 Rules .......................................................................................................... 14
  2.6 Message Routing ................................................................................................ 16
  2.7 Correctness Proof of HR Algorithm ................................................................... 18
  2.8 Complexity Analysis ......................................................................................... 24
CHAPTER 3 SELF-STABILIZING COMPACT ROUTING .............................................. 26
  3.1 Previous Work ..................................................................................................... 27
  3.2 Model and Notations .......................................................................................... 31
  3.3 Fault-Model .......................................................................................................... 29
  3.4 Compact Routing In Fundamental Topologies ................................................... 31
    3.4.1 Interval Routing Preliminaries .................................................................... 31
    3.4.2 Compact Routing Algorithm For 2-degree Bounded Graphs .................... 32
      3.4.2.1 Variables ............................................................................................ 33
      3.4.2.2 Labeling Module .............................................................................. 33
      3.4.2.3 Routing Module .............................................................................. 34
    3.4.3 Correctness Proof for 2-degree Bounded Algorithm .................................... 35
  3.4.4 Compact Routing Algorithm For Hypercubes .............................................. 38
    3.4.4.1 Variables and Macros .......................................................................... 40
3.4.4.2 Labeling Module ................................................................. 40
3.4.4.3 Routing Module ................................................................. 41
3.4.5 Correctness Proof for Hypercube Algorithm ......................... 41
3.5.2 Self-Stabilizing Compact Routing Algorithm For Arbitrary Topologies ........ 43
  3.5.2.1 Variables and Macros ...................................................... 44
  3.5.2.2 Labeling Module ............................................................. 44
  3.5.2.3 Routing Module ............................................................. 45
3.5.3 Proof Of Correctness for PR Algorithms ................................. 45
BIBLIOGRAPHY .............................................................................. 47
ACKNOWLEDGMENTS

My first thanks goes to my advisor Dr. Datta, who provided continual encouragement and interesting challenges to my work. He is simply an outstanding instructor, certainly one of the very best I've had in my college life, and my accomplishments in research are in large part tribute to his hard work, guidance, and advice. I would also like to thank Dr. Gewali, Dr. Taghva, and Dr. Iyer for serving on my committee. I would also like to thank the Computer Science Department for rewarding me with a graduate assistantship. Very special thanks goes to Dr. Yfantis, for his tireless efforts in helping me grow as a computer scientist, in sharing his profound wisdom, and in being a good friend. Additional thanks goes to Jerry Derby, his collaboration in the hierarchical routing chapter with me improved this work a great deal.

I would like to dedicate this thesis to my mother whose enduring support has made this thesis possible. I thank God am I so fortunate to have one who has sacrificed so much, and I cannot think of anyone in this world I admire more. I only hope someday I can say I gave back a tenth of what I have received.
CHAPTER 1

INTRODUCTION

Section 1.1  Distributed Systems

A distributed system is a interconnected collection of autonomous processes. These processes are able to communicate via either a shared memory or message passing mechanism. Algorithms written for distributed systems are called distributed algorithms, since they are ‘distributed’ both physically and concurrently over numerous processes.

Distributed systems have several features which differentiate them from sequential systems. By definition of a distributed system, each process is completely autonomous, therefore there is no centralized control. Additionally, processes have only a partial knowledge of the global topology and global state of the system. Lastly, synchronization of the network is allowed to fall into several different categories, typically the system may be either synchronous or asynchronous.

The use of distributed systems over sequential systems has great many advantages. These advantages are especially useful since communication networks continue to play a increasingly important and powerful role in today's society. These advantages include increased performance through better usage of system resources and increased reliability through replication. Additionally, distributed systems promote increased resource sharing and facilitate a more modular design of the network by increasing specialization. Of course, these advantages do not come without paying some price. The absence of central control, lack of global knowledge, and possible asynchrony introduce challenges into the design of distributed algorithms, making the distributed world inherently more complicated than sequential.
A sophisticated approach to the design of distributed algorithms includes considering the chance that nodes and links may fail. Because of the dispersion of processing resources in a distributed system, it is desirable to attain a partial failure property; no matter what failure occurs, only part of the system is affected. Fault-tolerant services greatly increase the flexibility of a system. Unfortunately, fault tolerance has often been difficult to achieve, due to the loosely connected nature of distributed computing.

One of the most inclusive and unified approaches to fault-tolerance in distributed systems design is the paradigm of Self-Stabilization [22]. A self-stabilizing system regardless of the initial state of the nodes and the initial messages in the links, is guaranteed to converge to the intended behavior within finite time. Thus the advantages of a self-stabilizing algorithm include an automatic recovery from illegitimate global states. Self-Stabilizing algorithms offer one of the most all-inclusive approaches to fault-tolerance and are useful protection against transient failures, i.e. temporary misbehavior of system components. A good survey of the self-stabilizing literature can be found in [32].

Many approaches to achieving self-stabilization exist. B. Awerbach, B. Patt-Shamir, and G. Varghese have given a method of decentralized detection and recovery in [18]. An alternative design can directed by closure and convergence. Convergence actions which move the system to the proper domain and closure actions which maintain the proper domain once achieved. Readers can refer to [16], [17] for details. A new paradigm called counter flushing in [35] was introduced by Varghese. Counter flushing achieves self-stabilization by synchronizing the system using time-stamped messages.

Fault-tolerance is an especially important issue in the design of network routing protocols since the topology changes due to the link/node failures and repairs. Several papers have been written in the area of self-stabilizing spanning tree construction [17,19,21,24,25]. Self-stabilizing shortest path problem has been studied in [36]. An optimal self-stabilizing
shortest path tree construction is presented in [15,33]. Self-Stabilizing topology update problem also got attention [19,23,26,30].

Section 1.3 The Routing Problem

Routing in distributed systems can be described as locally constructing at each processor \( i \) a uniform function \( \Gamma(i) \) s.t. given an input destination and a received message, the output image of \( \Gamma(i) \) is a suitable neighbor of processor \( i \) to forward the message to. The routing problem is one of the most fundamental in communication networks.

The quality of a routing scheme is measured by several factors. In order to reduce the time of delivery and maximize throughput of the system, a routing function that uses the least number of hops is highly desirable. Additionally, the topology of the network can be arbitrary, hence it is useful to consider universal schemes that apply to all network topologies. Lastly, the size of the network may become very large, thus it often becomes practical to consider compact routing methods. These compact routing schemes are capable of lower memory requirements independent of network size. So these schemes allow the network to be more scalable to possible future expansion. In summation, the quality of a routing function can be evaluated by its memory requirements, length of routing paths, and the extent of topologies to which it is well-defined.

Many researchers have achieved important results that have helped characterize the nature of the routing problem. It is additionally noted that currently there are a wide variety of routing algorithms in existence.

A classic solution to the routing problem is the use of all-pairs routing tables. The idea here is to store at each node an entry for each possible destination that indicates the next neighbor to forward the message to. The use of routing tables is an universal scheme that uses shortest paths, but requires \( O(n \log(n)) \) at each node.

All routing methods share the same property given by the example of all-pairs routing
tables. This property is that there exists some trade-off among the desirable qualities of any routing algorithm. The disadvantage of the high memory requirements of routing tables led to the development of ‘compact routing’ schemes that are efficient in memory requirements, very simple algorithmically and can be implemented with a very small amount of hardware in comparison with routing tables. However, these schemes are typically partial routing schemes that are not well-defined on all topologies. Additionally, compact routing schemes often use a path length that is greater than optimal. Awerbach, et. al. [13] introduced a very useful term for describing less than optimal paths, called stretch factor which is the maximum ratio between the cost of a route computed by a scheme and that of a cheapest path connecting the same pair of nodes.

It has been formally proven that universal schemes where relabeling of nodes is not allowed for a network of size \( n \) with maximum degree \( d \) require \( \Theta(n \log(d)) \) at each node [1]. Furthermore, it has been shown that for universal schemes and for any constant \( \epsilon : 0 < \epsilon < 1 \), \( \Omega(n \log(d)) \) is required locally for \( n^\epsilon \) routers even if the stretch factor is allowed to be at most two [2]. This result improved upon a previous result that states that \( \Omega(n^2) \) is required in total for routing schemes using length of path at most two times optimal [3].

Peleg and Upfal have also shown in [4] that for any stretch factor \( s : 2 < s < 16 \) requires \( \Omega(n^{s/16}) \). Therefore, if optimal path length of prime importance, it appears routing tables are the best solution for arbitrary topology networks where nodes may not be relabeled.

Hence, to construct a compact routing algorithm, we have three alternatives:

- Use a fixed topology.
- Settle for a stretch factor greater than 2.
- Assume we can relabel the nodes.

Typically, compact routing schemes do at least one of the above. This thesis will investigate each of these alternatives.
CHAPTER 2

SELF-STABILIZING HIERARCHICAL ROUTING

The hierarchical routing protocol represents a compromise between the high space complexity of an all-pairs shortest path routing algorithm [20] and a routing algorithm which does not support routing between all pairs of nodes or one which routes along a path with a distance much higher than optimal. This model has applications in important distributed computing areas such as ATM networks. The hierarchical model divides the network in portions, so that the space complexity required for the storage of the routing tables, cost of broadcast and other topological update tasks will be much more efficient. A segment of a node network layer address is devoted to which ‘portion’ of the network that the node belongs to, while another distinguishes destinations within that portion. As an example, the post office essentially uses hierarchical addresses. The first step to routing will be to get the letter to the correct country, per say to France. The next step will be to get the letter to the relevant district within France.

This chapter presents an algorithm of the well-known hierarchical routing model supporting fault-tolerance under the self-stabilization paradigm. Hierarchical routing model provides a less expensive algorithm compared to the traditional all-pairs routing algorithms. This algorithm benefits from the lower memory requirement, faster routing table lookup, and less costly broadcast exemplified by hierarchical routing and yet maintains routing capability of all pairs of connected nodes even in the presence of faults, such as link/node failures and repairs and and corruption of program variables. Additionally, this algorithm solves the problem of area partition where nodes that are supposed to be in the same subset of the network become isolated apart by link or node failure. Being self-stabilizing, starting from an arbitrary state (with possibly corrupted routing tables), the protocol is guaranteed
to reach a configuration with routing tables containing valid entries in a finite time. The protocol automatically updates the shortest paths in the face of dynamically changing link weights. The proposed protocol dynamically allocates/deallocates storage for the routing information as the network size changes. The algorithm works on an arbitrary topology and under a distributed daemon model.

Section 2.1 Previous Work

The hierarchical routing model is a compact routing algorithm first formally presented in [27,28]. Refinement of this model can be seen in [31]. The distributed hierarchical routing algorithms are presented in [27,28,31]. Lentfert, et. al. [29] defined a new distance metric, called hierarchical distance to present distributed hierarchical routing protocols. This distance metric leaves out the details of a path that are inside the domains of which the source node has no knowledge of. In earlier works [27,28], the distance is measured in terms of the least number of hops between two nodes. Awerbach, et. al. [13] introduced a very useful term, called stretch factor which is the maximum ratio between the cost of a route computed by a scheme (like the hierarchical routing protocol) and that of a cheapest path connecting the same pair of nodes. This paper introduces two families of routing protocols, called hierarchical covering pivots and hierarchical balanced schemes.

None of the above-mentioned papers on hierarchical protocols is self-stabilizing. The algorithm presented in this paper is a self-stabilizing implementation of the hierarchical routing model. This algorithm will support faults causing node and link failures and additions. Additionally, faults which cause area-partitions (c.f. Section 2.4.1) are allowed by this algorithm.
Section 2.2 Model and Notations

A (distributed system) network of $n$ nodes can be modeled as a graph $G(V,E)$, where $V = \{v_1, v_2, v_3, \ldots, v_n\}$ is the set of nodes (or vertices) in the network and $E$ is the set of edges (links). At any instant, each node is either up or down. The set of up nodes is maintained by an underlying protocol.

Each node has an unique identifier. To facilitate a m-level hierarchically divided graph, identifiers are defined as a $(m+1)$-tuple of the form $<area_id_m, area_id_1, area_id_0>$. $area_id_0$ indicates the localmost id. Each node $i$ can be thought of as a singular area composed of only itself. For notational convenience, the highest level, level $m$, is a singular area that all nodes belong to. $i.Area[i]$ is used to indicate the $area_id_i$ at node $i$. The model supports composite atomicity so that a node can read the value of its and its neighbors' variables and writes its registers in a single atomic step. The asynchrony of the system by introducing a distributed daemon execution model; if a distributed daemon is present, at any time, any subset of the set of privileged processes may move.

The algorithm executed by a process has the following form:

$$\langle\text{rule}\rangle$$

$$:\quad :\quad :$$

$$\langle\text{rule}\rangle$$

Each rule has the form:

$$\langle\text{guard}\rangle \rightarrow \langle\text{action}\rangle$$

A guard is a boolean expression over the variables owned by the node and those of its neighbors. An action results in an assignment to one or more of the variables owned by the node executing the action. When, in a node $i$, one or more guards are satisfied, $i$ non-deterministically executes one of the corresponding actions; as written by Dijkstra [22], that node enjoys or has a privilege and may make a move.

A local state of processor $i$ is defined as a description of the variables of $i$. A global state
is the set of all local states. A computation step is the atomic execution of an action at a single processor. A computation is a sequence of computation steps.

A problem specification $SP$ is a description of objectives. A legitimacy predicate $LP$ for protocol $P$ is a list of boolean predicates specified with respect to global state of $P$, that when satisfied, invariantly satisfies $SP$ in all computations of $P$. Protocol $P$ is said to be self-stabilizing if the following two conditions hold invariantly:

Convergence: Starting from an arbitrary global state, any computation of $P$ reaches a global state such that $LP$ is satisfied within a finite amount of time.

Closure: Any computation step taken while $LP$ is satisfied leaves $LP$ satisfied.

Section 2.3 Hierarchical Routing

The problem solved in this chapter is to compute a hierarchical routing table at each node which will eventually have valid routing information: the tables will not contain the information about the unreachable nodes in the network; the tables will contain the shortest distance to a destination node and the id of the next node in the shortest path to the destination. The algorithm will handle the area partitioning problem (see Section 2.3.1) and automatically adjust to the change of topology and link weights.

The algorithm presented in this paper will generate a routing table using the hierarchical routing model. The hierarchical routing model provides a compromise between the optimal routing and minimal storage space. A primary objective of this model is to reduce the amount of routing information computed and stored at each node while maintaining a near-optimal routing scheme which will allow routing messages between all pairs of nodes in the network.

The hierarchical scheme groups the nodes of the network into areas. Each area is a subset of the nodes of the network. Every node belongs to exactly $m$ areas in a hierarchi-
cal network. In a fault-free network, all areas are internally connected. The fundamental components that define a hierarchically divided network are defined below:

**Definition 2.1**  
Nodes $i$ and $k$ are $l$-similar nodes if $\{\forall q : q \geq l : i.\text{Area}[q] = k.\text{Area}[q]\}$.

**Definition 2.2**  
A link $(i,j)$ is an $l$-internal link if and only if $i$ and $j$ are $l$-similar nodes. 

A path is an $l$-internal path iff all links in the path are $l$-internal.

As already observed, the general strategy of hierarchical routing is for each node to store a proper subset of all nodes. To meet this objective, each superset of $l$-internally connected components is treated at level $l + 1$ as single node. Additionally, at each level $l$ of the area hierarchy, each node $i$ only maintains a routing table entry for each superset of $(l-1)$-similar nodes (called a desirable set) that are on a $l$-internal path from $i$. For example, node $i$ stores all level 0 similar nodes (each superset being a single node at most) reachable via only level 1 internal links, all level 1 supersets reachable via only level 2 internal links, etc. Details that are embedded inside a desirable set are effectively hidden from node $i$.

Note a hierarchical path between two nodes in a hierarchically divided graph is well-defined if it uses a series of non-increasing $l$-internal layers, i.e. each link that is $l$-internal to the destination is followed by a $l_0$-internal link with $l_0 \leq l$.

Routing messages in a hierarchically divided network is complicated by the fact that a single table entry may represent several nodes. To send a message between nodes, the source routes using the lowest level $l$ in the hierarchy such that the destination and source are $l$-similar. For example, if node $s = <I,A,e,10>$ wanted to send to $<I,A,f,44>$ then level 2 (id A) would be used to send from area $e$ to the closest node $k$ in area $f$ from node $s$. Node $k$ will then use a $l$-internal path to route to the destination. Thus, it is possible to route messages between all pairs of connected nodes in a hierarchical network without storing all nodes at each destination.
Section 2.4 Fault-Tolerant Hierarchical Routing

The algorithm presented in this thesis will generate an hierarchical routing table at each node which will eventually (in a finite time) have valid routing information in spite of transient errors. Faults may result in a situation in which an area is partitioned. This situation is defined and discussed in detail in Section 2.4.1.

Since the algorithm supports a changing network configuration it requires the use of dynamically allocated memory. This is discussed in Section 2.4.2. The self-stabilizing hierarchical routing algorithm is divided into two modules. The algorithm to create and maintain the hierarchical routing tables is given in Section 2.5. Finally, Section 2.6 describes how the enhanced routing tables generated by the table maintenance can be used to route messages.

Section 2.4.1 Area Partitioning

Since link and node failures are permitted in the network, the algorithm must consider the case where l-similar nodes are not l-internally connected. If this occurs, the area is said to be partitioned. An area which has been partitioned can be said to be divided into sub-areas. The l-sub-area to which a node belongs is the set of all nodes, including itself, which are reachable using only l-internal links. Using this definition, it is clear that an unpartitioned area is also a sub-area. A more formal definition of a sub-area is given below:

Definition 2.3 For every node i in the network, the l-sub-area of i (denoted by S_A(i)[l]) is the set of all nodes, including i, connected to i by an l-internal path (c.f. Definition 2.2). This set can be recursively defined as:

\[ S_A(i)[l] = \{i\} \]

\[ S_A(i)[l] = S_A(i)[l] \cup \{x \in V|x.Area[l] = i.Area[l] \land (\exists y \in S_A(i)[l])((x,y) \in E)\} \]
The algorithm presented in this paper attempts to route messages in the event of failures which cause an area partition. The fault-tolerant routing table at each node stores one entry for each l-sub-area in the network that is a desirable set, which includes sub-areas of partitioned areas: namely those desirable sets that are l-similar and yet have the same (l-1) area number. As an additional example, if a level 2 area which we'll call $B$ becomes partitioned into two sub-areas then level 3 will maintain a path between the both of them if possible. If level 3 cannot do so, then level 3 must also be partitioned. All nodes belong to level $m$, so at some level in the hierarchical structure a path knowledge is maintained between all nodes of area $B$. Of course at whatever level $l$ that is, both sub-areas will be both $l$ and $(l - 1)$ similar. In a fault-tolerant system it is easy for such a situation of several l-sub-areas to exist each having the area id $l - 1$. To avoid a possible ambiguity, the algorithm must be able to distinguish between these sub-areas. This can be accomplished by exploiting the fact that every node has a unique identifier. The table maintenance algorithm elects a leader for each sub-area in the network, and the identifier of the leader is used as the sub-area identifier.

Section 2.4.2 Dynamic Memory Allocation

The algorithm presented in this paper will support events in which nodes are added to or removed from the network. At each node, the algorithm stores routing information for every node within the same sub-area and for every sub-area within the network. This gives a state space of the $O(n)$, where $n$ is the number of node in the network. Since $n$ may change during algorithm execution, the state space of the algorithm cannot be fixed in advance. Thus the algorithm must be able to dynamically allocate additional storage for the routing information as the network size increases.

Due to the dynamic allocation of memory, cases will arise where routing information for an recently added node has not been stored in the routing table. This can complicate the
presentation of the code, if the code must always determine if a routing table entry exists before it can be examined or modified by the algorithm. However, this complication is not significant to the functioning of the algorithm and it can be abstracted out with two simple assumptions which give a simple notational convention which allows clearer presentation of the code of the routing table maintenance algorithm.

Assumption 2.1 If an entry does not exist in the routing table, and the code attempts to make an assignment to that non-existent entry, then the entry is created in the routing table before the assignment is made.

Assumption 2.2 If an entry does not exist in the routing table, and the code attempts to read the value stored in that entry, then the apparent value read will be $e$. The only necessary property for $e$ is that $e$ cannot be an element of the domain of the variable being read.

The use of these two assumptions can be justified by the fact that they will allow simplification of the notation when writing the rules. Without these assumptions, many rules in the code would need to be split into multiple rules and the rules would be more complex. This can be shown by comparing two versions of a code fragment used in the routing table maintenance algorithm. One version of the code which does not use the assumptions and one version which uses the assumptions are given below. The code fragment shown below stabilizes a node's distance to itself (stored in the variable $Dst_i(i)[i]$) to zero.

If the code presented in this paper does not use Assumptions 2.1 and 2.2, then the code must first verify the existence of the variable $Dst_i(i)[i]$ before reading it or writing to it. For this purpose the function $TABLE(Dst_i(i)[i])$ is defined to return true if and only if the variable $Dst_i(i)[i]$ exists in the internal routing table. The code fragment is shown below:

```c
/* Create and stabilize the distance value for self */
```
RIA :: \sim \textsc{table}(\text{dst}_i[0]) \rightarrow
\text{Create dst}_i in routing table;
\text{dst}_i[0] := 0;

/* Stabilize the distance value for self */
RIB :: \textsc{table}(\text{dst}_i[0]) \wedge (\text{dst}_i[0] \neq 0) \rightarrow
\text{dst}_i[0] := 0;

With the Assumptions 2.1 and 2.2, the code fragment can be expressed in a more concise form. This version of the code fragment can be written as a single rule which is simpler than the code given above. This code is shown below:

/* Stabilize the distance value for self */
R1 :: (\text{dst}_i[0] \neq 0) \rightarrow
\text{dst}_i[0] := 0;

Assumptions 2.1 and 2.2 will be used hereforth without further discussion.

Section 2.5 Routing Table Maintenance Algorithm

This section gives the code for generating and maintaining the routing tables for the fault-tolerant hierarchical routing algorithm. The routing tables store the distance for each entry in the table and an entry which stores the id of a best neighbor to use when routing a message to that destination. Additionally, each level elects a leader of each sub-area A leader at level \(l\) provides a destination label for \((l+1)\) similar nodes to connect to.

A nice feature of our routing table maintenance algorithms is that link weights can be changed dynamically and the algorithm will stabilize using local checking and local
correction [18, 34].

The notation $Max\_Diam[l]$ indicates a limit of the maximum path cost of an l-
internal path in the network at any time. This constant is needed to allow the algorithm to detect node failures and allows invalid entries to be removed from the routing tables (this is similar to using a timeout to detect link failures).

Two sets are defined below to simplify presentation of the algorithm. It is worth noting that the set must be computed each time a guard is evaluated or an action is executed if the set is referred to in that rule. The set $R\_Nbrs_i(k)[l]$ is the set of neighbors of $i$ which are l-
similar to $i$ and which have an entry in their l-level routing table which stores the distance to node $k$. Notice that the set $R\_Nbrs_i(k)[l]$ (see below) only contains nodes which are in the same area as $i$ and which are connected by an l-
internal link. By Definition 2.3 it can be seen that $R\_Nbrs_i(k)[l]$ is a subset of the sub-area of $i$ ($S\_A(i)[l]$). The set $Min\_Nbrs_i(k)[l]$ is the subset of nodes in $R\_Nbrs_i(k)[l]$ which are on a minimal l-
internal path from node $i$ to $S\_A(k)[l]$ that is less than or equal to $Max\_Diam[l]$. The variable $Dst_i(k)[l]$ stores the distance of the shortest l-
internal path from node $i$ to $S\_A(k)[l]$. $Nxt_i(k)[l]$ stores the node id of a l-
similar neighbor of $i$ which is on a shortest l-
internal path from node $i$ to $S\_A(k)[l]$.

Section 2.5.1 Variables and Macros

$$R\_Nbrs_i(k)[l] = \{x : (x \in Nbrs_i) \land (Dst_x(k)[l] \neq e) \land (k.Area[l] = i.Area[l])\}$$

$$Min\_Nbrs(k)[l] = \{x \in R\_Nbrs(k)[l] : ((Dst_x(k)[l] + wt(i, x)) \leq Max\_Diam[l]) \land ((Dst_x(k)[l] + wt(i, x)) = \min_{y \in R\_Nbrs(k)[l]} (Dst_y(k)[l] + wt(i, y)))\}$$

$$LDR_i(l)\{$$

if ($l = 0$)

return $i$;
Section 2.5.2 Algorithm Rules

parameter \( l \) \( 1 \ldots m \)

/* Elect a leader for the sub-area and store in \( Ldr_i \) */
\[ R0 :: Ldr_i[l] \neq \min \{ x \in V : (Dst_i(x)[l] \neq e) \land (x.Area[l] = i.Area[l]) \} \rightarrow Ldr_i[l] = \min \{ x \in V : (Dst_i(x)[l] \neq e) \land (x.Area[l] = i.Area[l]) \} \]

/* Stabilize the distance value for self */
\[ R1 :: Dst_i(LDR(l-1))[l] \neq 0 \rightarrow Dst_i(LDR(l-1))[l] := 0; \]

/* Stabilize the next variable for self */
\[ R2 :: Nxt_i(LDR(l-1))[l] \neq i \rightarrow Nxt_i(LDR(l-1))[l] := i; \]

/* Update the distance value */
\[ R3 :: (\exists k : k \neq LDR(l-1)) : (\exists j \in Min.Nbrs_i(k)[l]) \land (Dst_i(k)[l] \neq (Dst_j(k)[l] + wt(i,j))) \rightarrow Dst_i(k)[l] := Dst_j(k)[l] + wt(i,j); \]

/* Update best neighbor from \( i \) to \( k \) */
\[ R4 :: (\exists k : k \neq LDR(l-1)) : (\exists j \in Min.Nbrs_i(k)[l]) \land (Nxt_i(k)[l] \notin Min.Nbrs_i(k)[l]) \rightarrow Nxt_i(k)[l] := j; \]

/* Remove the distance entry for an invalid node \( k \) */
\( \exists k : k \neq LDR(l - 1) : Dst_i(k)[l] \neq \epsilon \) and \( (\text{Min}_Nbrs_k)[l] = \phi \) →

Remove \( Dst_i(k)[l] \) from the routing table;

\( /* \text{Remove the best neighbor entry for an invalid node } k */ \)
\( \exists k : k \neq LDR(l - 1) : Nxt_i(k)[l] \neq \epsilon \) and \( (\text{Min}_Nbrs_k)[l] = \phi \) →

Remove \( Nxt_i(k)[l] \) from the routing table;

The rules given above can be described informally. Rule R0 elects the node in the sub-area of \( i \) with the lowest id as the leader of that sub-area. Rule R1 ensures that the node always shows the distance to its own l-sub-area to be zero. Rule R2 fixes the next best neighbor from \( i \) to own l-sub-area to be itself. It should be understood that \( k \) is used as a dynamic label for \( S.A(k)[l] \) and does not necessarily indicate the actual node id of the node that the best hierarchical path from \( i \) connects to. It is also important to note that although the choice of \( k \) is non-deterministic, \( k \) always corresponds to an entry that is already in a routing table. Rule R3 updates the distance from \( i \) to \( k \) when the current value of \( Dst_i(k)[l] \) differs from the best distance. Rule R4 updates best neighbor pointer for a path from \( i \) to \( Nxt_i(k)[l] \) when the current best neighbor is not on a minimal l-internal path from \( i \) to \( S.A(k)[l] \). Rules R5 and R6 remove the routing entries for node \( k \) if there is no l-internal path from node \( i \) to \( S.A(k)[l] \).

Section 2.6 Message Routing

The algorithm presented in this paper will generate a fault-tolerant hierarchical routing table. Since the algorithm supports area partitioning, the standard algorithm for routing using an hierarchical routing table is insufficient when attempting to route messages in the event of an area partition.

The routing algorithm presented here assumes that the routing tables have already
stabilized to correct values. If the routing tables have not stabilized, then the values of the routing tables are not clearly defined and it cannot be guaranteed that proper routing will occur.

A node can determine if an area is partitioned by examining the routing table. The routing table will store one entry for every desirable set in the network. If the routing table contains exactly one sub-area entry for an area, then the area is not partitioned, otherwise more than one entry in the routing table for a single area indicates that that area is partitioned.

Consider a case where a node \( i \) which wishes to send a message to a node \( k \). If node \( k \) is in a partitioned area \( A \), then \( i \) cannot know which sub-area (if any) contains node \( k \). The best choice for node \( i \) is to send a message to every sub-area which might contain node \( k \) (every sub-area with a leader similar to node \( k \)). If the area of node \( k \) is not partitioned, then a single message is sent to the sole sub-area with a leader similar to node \( k \).

Pseudo code is given below to show how messages are routed using the fault-tolerant hierarchical routing tables. The first code segment given below shows how a node \( i \), which originates a message \( msg \) for destination node \( k \), will route the message. Following this is code showing how a message is forwarded along a hierarchical routing path.

It is noted that the routing algorithm presented here could conceivably result in a high message complexity. However, for simplicity of presentation and lack of space, this issue is simply ignored. A more message efficient algorithm can conditionally check if the neighbor the message was received from sent the message at the same level the current node is intending to send the message at. If not, no new information at this node exists concerning area partitions and only one message needs to be sent regardless of the status of area partitions in the network.

\[ L_Set_i(k) = \text{the set of table entries at node } i \text{ minimally } l\text{-similar to } k. \]
Receive message \((k, msg)\)

If \(L.Set(k, l) = \phi\) 

\[ \text{Nack/Discard } msg; \]

Else If \((k = i)\)

\[ \text{Deliver } msg; \]

Else

\[ \forall k \in L.Set(k): \]

Let \(l\) be of minimal value s.t. \(\text{Nxt}(k)[l] \neq \phi\)

\[ \text{Send } (k, msg) \text{ to } \text{Nxt}(k)[l]; \]

A lower layer in the network protocol handles verification of message delivery. The statement Nack/Discard is written when a node \(k\) decides that the message cannot be delivered with the current addressing and routing information at node \(k\). The verification performed by the lower layer might send a negative acknowledgment, discard the message, or perform some other action; however this is beyond the scope of this thesis. The routing algorithm can be adapted to support a mixed hierarchical and non-hierarchical network. If the network is mixed, then messages solely remaining within the hierarchical or non-hierarchical networks are routed using standard routing mechanisms. If a message passes between networks, then it must pass through a bridge such that the node on one side of the link is in the hierarchical network and the node on the other side of the link is in the non-hierarchical network. These bridge nodes in both networks must have additional instructions defining how address translation must occur to pass messages from one network to another. If this is true, then bridge node needs only to translate the message and then pass it across the bridge, where it will be routed normally by the network on the other side of the bridge.
Section 2.6 Proofs of Correctness of HR Algorithm

The problem has been informally in Section 2.3. For the proofs of correctness, a formal definition, is here constructed.

Definition 2.4 The hierarchical routing problem is to satisfy the following:

Each node \( i \in V \) has a table entry describing a minimal \( l \)-internal path to \( l \)-sub-area \( k \) iff \( k \) is reachable using a \( l \)-internal path from \( S.A(i)[l] \).

The next step is to define a correct global state of the network. In doing so, it is argued if the predicate \( \mathcal{L}_T \) is satisfied in finite time in all possible computations then the algorithm presented in this thesis solves the hierarchical routing problem.

\[
\begin{align*}
I_1 & \equiv \{ \forall i \in V, \forall l : Dst_i(LDRI[l - 1])[l] = 0 \land Nxt_i(LDRI[l - 1])[l] = LDRI[l - 1] \} \\
I_2 & \equiv \{ \forall i \in V, \forall k \in S.A(i)[l] : k \neq LDRI[l - 1] : \\
& \quad (\exists j : j \in Min.Nbrs_i(k)[l] : Dst_i(k)[l] = Dst_j(k)[l] + wt(i, j) \land Nxt_i(k)[l] = j) \\
I_3 & \equiv \{ \forall i \in V, \forall k \notin S.A(i)[l] : Dst_i(k)[l] = \epsilon \land Nxt_i(k)[l] = \epsilon \\
I_4 & \equiv \{ \forall i, k \in V, \forall l : S.A(i)[l] = S.A(k)[l] \iff ldri[l] = ldri_k[l] \}
\end{align*}
\]

\( \mathcal{L}_T \equiv I_1 \land I_2 \land I_3 \land I_4 \)

It will proven that \( \mathcal{L}_T \) stabilizes by proving that each invariant stabilizes individually. The definitions for the preceding invariants are justified by mentioning a minimum number of intuitively disjoint cases. The distance and next variables both are either the special value \( \epsilon \) or are correct values unequal to \( \epsilon \) indicating a minimum \( l \)-internal path to \( k \). The leader variable of level \( l \) has self-stabilized if it is the same for each node in the same \( l \)-sub-area. The distance and next values are to the self or they are not, and need to be defined differently in each case.

The invariants can be informally described as follows: \( I_1 \) is interpreted as meaning a node's distance to its own area is always 0 and \( i \) stores itself as the next its hierarchical
path. \( I_2 \) indicates that if there exists an \( l \)-internal path from \( i \) to \( k \), both \( Dst_i(k)[l] \) and \( \text{Next}_i(k)[l] \) contain valid entries. The meaning of \( I_3 \) is that if \( k \) and \( i \) are not in the same \( l \)-sub-area, node \( i \) does not keep \( k \) in its routing table. \( I_4 \) is interpreted as if two nodes \( i \) and \( k \) are in the same \( l \)-sub-area if and only if they have the same node id as their sub-area leader.

First the proof is motivated by the following discussion. The proof of the algorithm is intuitively a proof on the number of levels. The proof of the basis step, however, depends on whether or not any execution sequence that contains only levels \( 2 \) through \( m \) is always a finite computation. If that is left unproven before the basis step is completed, it can always be claimed that level \( 1 \) starts from an illegitimate state and never converges because the daemon never schedules it, infinitely choosing some subset of the levels \( 2 \) through \( m \).

With the preceding paragraph in mind, the outline of the proof is as follows. In simplistic terms, the first step is to let \( m_0 \) equal the lowest level is that is never forced out of the execution sequence by a scheduler running a non-terminating computation. More specifically, let \( m_0 \) equal the lowest level that appears infinitely often at any processor that appears infinitely often in the computation of the algorithm. If the computation of the algorithm is finite, then \( m_0 \) is the lowest level that appears in the computation. There must exist a \( m_0 \) since number of levels is finite. It is first proven that \( m_0 \) stabilizes for level \( m_0 \) only for each of the four invariants. Then it can easily be argued that the choice of \( m_0 \) is without loss of generality, in the final, inductive proof of the algorithm.

As previously noted, the details of lower levels are hidden from the higher levels. Level \( m_0 - 1 \) influences \( m_0 \) only by providing a set of possible destinations for \( m_0 \) (the set of leaders). For level \( 1 \), it should be intuitively clear that this set is only nodes that are in the network, others cannot hold a distance value of \( 0 \) in the network to avoid the count up to Max_Diam limit. This will proven formally shortly. So if all levels less than some \( m_0 \) between \( 1 \) and \( m \) eventually stop (or possibly no such levels), it follows of course \( m_0 \)
only considers a limited set of destinations already in some routing table in the network or actually in the network and nothing else. Since no level less than \( m_0 \) is in the computation the set of destinations has essentially settled. Note the case when a node creates its own routing table entry is clearly closed. So the algorithm, if correct, will run in a similar way to a 1-level all-pairs shortest paths algorithm with the only difference being that several paths can be constructed to the same destination sub-area.

Once it has been proven that the algorithm stabilizes for level \( m_0 \), we construct an iterative proof from \( m_0 \) to \( m \) using the same logic that shows that the computation must always be finite. It is possible to construct a final inductive proof from 1 to \( m \) that the algorithm is self-stabilizing.

Lemma 2.1 The set of states \( I_1 \) is self-stabilizing under system execution.

Proof: Consider arbitrary node \( i \in V \). Only rule R1 and R2 can be enabled at node \( i \), for \( i = LDR[m_0 - 1] \). Thus, \( I_1 \) is closed and converges. □

Lemma 2.2 The set of states \( I_2 \) is closed under system execution.

Proof: The obligation is to show that for each state \( s \) in \( I_2 \) and for each action enabled at \( s \), executing the assignment statement of the action in \( s \) yields a state in \( I_2 \). This obligation is met as follows.

Assume the system is in state s.t. \( \exists j \in Min_Nbrs_i(k)[m_0] : (Dst_i(k)[m_0] = Dst_j(k)[m_0]+wt(i,j)) \land (Nxt_i(k)[m_0] = j) \). Note that only execution of R5 and R6 can violate \( I_2 \). Observe \( Min_Nbrs_i(k)[m_0] \neq \phi \), thus no rule is enabled at \( i \), and thus \( I_2 \) is closed. □

The invariants are defined such that the set of nodes is divided into two cases. A destination node is either in the same subnetwork as the source or it is not. This is done because the definition of \( Dst \) in these two cases is unrelated. It is proven that a node properly stabilizes hierarchical routing paths that they should have first. Then it is proven
that a node does not have entries that they are not interested in. These are disjoint
cases that do not affect one another. It should be mentioned that one cannot stabilize if it
is infinitely working on the other. It will be shown that both stabilize, therefore, neither
case could possibly be starved by a unfair scheduler.

The proof of $I_2$ is, as expected, an inductive proof on the number of hops from desti-
nation sub-area $k$ to each node in $S.A(k)[m_0]$. The proof is simple, but first the definition
given next must be understood.

Definition 2.5 Let $P(k)$ for destination $k$ equal the set of nodes in $S.A(k)[m_0]$ for which
1. $I_2$ holds
2. every distance value in $P(k)$ is less than any distance in $S.A(k)[m_0] \setminus P(k)$.

Lemma 2.3 $I_2$ converges in finite time.

Proof: The proof is an inductive argument starting from $k$ on the number of hops as follows:

\[ \forall i \in S.A(k), \text{ there exists an internal path between } i \text{ and } k \text{ by Definition 2.3. Let } x \text{ be not in } P(k) \text{ but adjacent to } P(k). \text{ R3 or R4 must be enabled at } x \text{ if } I_2 \text{ is not satisfied by definition of } P. \text{ By an induction hypothesis, } I_2 \text{ converges.} \]

The proof of convergence $I_3$ hinges on demonstrating that an entry $k$ not in the same
m-sub-area as $i$ will be forced by continual execution of R3 to increase past $Max\_Diam[m_0]$.

Definition 2.6 Let $M(k)$ equal the set of nodes having table entry $k$ that is not in its
$m_0$-sub-area, in its routing table.

Definition 2.7 Let $MVal(k)$ equal the minimum value of distance values taken over $M(k)$. 
Lemma 2.4 $I_3$ converges.

Proof: If $i$ and $k$ are not $m_0$-similar then clearly we are done, as only $R_5$ or $R_6$ can be enabled. First note that $MVal(k)$ cannot decrease because link weights are positive. Next observe that for arbitrary node $x \in M(k)$ there are two cases.

case 1: $\minNbrs_x(k)[m_0] \neq \phi$

$x$ executes $R_3$ or $R_4$ and $MVal(k)$ remains non-decreasing.

case 2: $\minNbrs_x(k)[m_0] = \phi$

$x$ executes $R_5$ or $R_6$. $MVal(k)$ remains non-decreasing.

Since the two cases apply to each node in $M(k)$, either $MVal(k)$ remains increasing or $M(k)$ becomes equal to $\phi$. Thus, $MVal(k)$ increases to maximum possible without exceeding $Max.\ Diam[m_0]$ unless all nodes have removed $k$ from their routing table. Eventually then each node in $S.A(i)$ has $\minNbrs_i(k) = \phi$. Only $R_5$ or $R_6$ is left enabled for these nodes. □

Lemma 2.5 The set of states $I_3$ is closed under system execution.

Proof: Assume that the system is in a state $s_0$ in $I_3$ where $(\forall k : ((k \in S.A(i)[m_0]) \land (\forall x \in S.A(i)[m_0] : Dst_x(k)[m_0] \neq \epsilon))$. Then clearly $\forall x \in S.A(i)[m_0] : \minNbrs_i(k)[m_0] \neq \phi$ unless $i = LDR_i(k)[m_0]$. Thus $R_5$ or $R_6$ cannot be enabled.

Note that $R_5$ and $R_6$ are the only rules which can remove an entry from the routing table of $i$. So, the entry for $k$ will not be removed from the table of $i$. By Definitions 2.2 and 2.3, $k$ remains in the sub-area $S.A(i)[m_0]$.

Next assume that the system is in a state $s_1$ in $I_3$ where $(\forall k, \forall i, \forall x \in S.A(i)[m_0] : ((k \notin S.A(i)[m_0]) \land (Dst_x(k)[m_0] \neq \epsilon)) \implies (\minNbrs_i(k)[m_0] = \phi)$. Therefore $R_3$ is not enabled.

Since $R_3$ is the only rule which can add an entry $(Dst_i(k)[m_0])$ for $k$ in the table of $i$, the entry for $Dst_i(k)[m_0]$ will not be made and $k$ remains outside the sub-area $S.A(i)[m_0]$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The analysis of the $\text{Next}_t(k)[m_0]$ variable follows the same pattern.

Definition 2.8

$$s_3 \equiv (\forall k \in V : ((S.A(i)[m_0] = S.A(k)[m_0]) \land (Ldr_1[m_0] = Ldr_k[m_0])))$$

$$s_4 \equiv (\forall k \in V : ((S.A(i)[m_0] \neq S.A(k)[m_0]) \land (Ldr_1[m_0] \neq Ldr_k[m_0])))$$

Lemma 2.6 The set of states $I_4$ is closed under system execution.

Note that the $ldr$ variables are modified only by the rule R0. Assume that the system is in a state $s_3$. By $I_2$, $(\forall k \in V : S.A(i)[m_0] = S.A(k)[m_0])$ implies $(\min\{x \in S.A(k)[m_0] : Dst(x)[m_0] \neq \epsilon\} = \min\{x \in S.A(i)[m_0] : Dst_k(x)[m_0] \neq \epsilon\})$. Therefore the system remains in $s_3[m_0]$.

Next assume the system is in state $s_4$, $(\forall k \in V : (S.A(i)[m_0] \neq S.A(k)[m_0]))$ implies $(\min\{x \in S.A(k)[m_0] : Dst_i(x)[m_0] \neq \epsilon\} \neq \min\{x \in S.A(i)[m_0] : Dst_k(x)[m_0] \neq \epsilon\})$ since id’s are guaranteed to be unique. The system remains in $s_4$.

Lemma 2.7 Upon starting from an arbitrary state, the system reaches a state in $I_4$.

Proof: It will be proven that starting from a state not equal to $s_3$, the system will eventually reach the state $s_3 \in I_4$. We will then do the same for $s_4$.

Assume that in the current state $(\exists i, k \in V : (S.A(i)[m_0] = S.A(k)[m_0]) \land (Ldr_i[m_0] \neq Ldr_k[m_0]))$. Observe that the system is not in $s_3$. Given that $I_1$, $I_2$, and $I_3$ stabilize, for any $x$ in $S.A(i)[m_0]$, $(Dst_i(x)[m_0] \neq \epsilon) \land (Dst_k(x)[m_0] \neq \epsilon)$, Therefore, $(\forall x \in V : \min\{x \in V : Dst_i(x)[m_0] \neq \epsilon\} = \min\{x \in V : Dst_k(x)[m_0] \neq \epsilon\}$. Hence, eventually $(Ldr_i[m_0] = Ldr_k[m_0])$. So, the system is in state $s_3$.

Assume the system is not in state $s_4$. Because $(S.A(i)[m_0] \neq S.A(k)[m_0])$ it follows from $I_1$, $I_2$, and $I_3$ that $\min\{x \in V : Dst_i(x)[m_0] \neq \epsilon\} \neq \min\{x \in V : Dst_k(x)[m_0] \neq \epsilon\}$. since id’s are unique. Eventually then we reach a state in $s_4$. $\square$
Theorem 2.1 {Closure of $\mathcal{L}$} The set of states $\mathcal{L}$ is self-stabilizing under system execution.

Proof: $I_1$, $I_2$, $I_3$ and $I_4$ are self-stabilizing, thus it follows that level $m_0$ is self-stabilizing. It is easy then to construct an iterative proof from $m_0$ to $m$ which shows that the computation sequence must always be finite, by definition of $m_0$. An simplistic indication that $\mathcal{L}$ is self-stabilizing is that $m_0$ must equal 1, otherwise level 1 simply would not converge, which it is obviously does given the arguments provided in the previous lemmas. The proof is given by a inductive hypothesis on the number of levels, thus $\mathcal{L}$ is self-stabilizing. □

Section 2.6 Complexity Analysis

The space complexity improvement of self-stabilizing hierarchical routing over non-hierarchical routing at a node is dependent on the definition of $m$ and the area hierarchy, which will symbolized by $A*$. The best and worst case complexity are described below.

Following the space complexity analysis is a discussion of the time complexity needed by the algorithm to converge to a legitimate state.

The space complexity at node $i \in V$ is the total number of nodes on an $l$-internal path from $i$ for $l \in 1 \ldots m$. In the best case at single node, a node may only belong to 1 level with the other $n - 1$ nodes being an additional area. In this instance a node may only need constant number of table entries, but the stretch factor in this case would likely be very high. The worst case at a node results for a node that belongs to $m$ levels, results when $m = n$ and $\forall k \in 1 \ldots m : |A_{k-1}| = (|A_k|) - 1$. yielding a complexity of $O(n^2(\log_2(n)))$ summing up the storage needed for each of the levels, which is considerably worse than the all pairs algorithm. The total average of the nodes in this case, does no better or worse, with $O(n \log_2(n))$.

The optimal space complexity results when $m = \log_2(n)$ and $|A_{l-1}| = (|A_l|)/2$, for $l \in 1 \ldots m$. Node identifiers can be stored in $\log n$ bits, so in this case the space complexity yields $(\log_2(n))^3$, since each level requires $\log_2(n)$ table entries.
The analysis of these cases suggests that the best way to design $A^*$ is in a balanced way.

A drawback of this algorithm, is that in the worst case (worst definition of $A^*$), the stretch factor is in fact unbounded. It is unbounded because it depends upon $n$, the number of nodes in the network, which can continually grow.

The time complexity of the algorithm can be determined by examining the longest delay (counted by the distance between nodes) before all pertinent information at a node becomes correct. The time complexity is the same as the non-fault tolerant algorithms referenced.

$Diameter[l]$ is the largest distance between any node in the network and the closest node in the most distant sub-area[$l$]. Each node $i$ will update the $Dst_i(k)[l]$ and $Nxt_i(k)[l]$ entries for all nodes $k \in S_A(i)[l]$ in $O(Diameter[l])$ time. All invalid entries will be removed from the $l$-level routing table in $O(Max_{Diam}[l])$ time.

As previously noted, it must be assumed that $Max_{Diam}[l] \geq Diameter[l]$. Thus, an $l$-level routing table will stabilize in $O(Max_{Diam}[l])$ time and the algorithm will stabilize in $O(\sum_{l=1}^{m} Max_{Diam}[l])$ time.

The self-stabilizing hierarchical routing algorithm presented here will converge to a valid state in $O(\sum_{l=1}^{m} Max_{Diam}[l])$, if no faults occur. Since all $Diameter[l]$ are $O(n)$, the will have the time complexity will be $O(mn)$ in a fault-free network.
CHAPTER 3

SELF-STABILIZING COMPACT ROUTING

This chapter presents several compact routing algorithms for fault-tolerant distributed systems using the paradigm of self-stabilization. Self-stabilization guarantees to eventually satisfy an legitimacy predicate given an arbitrary fault in the system. The proposed algorithms are simple, intuitive and can efficiently implemented with a small amount of hardware.

Section 3.1 Previous Work

Given a set of destinations and a set of ports, a compact routing scheme reduces the size of the routing table at each node to less than \( \Theta(n \log(d)) \), where \( d \) is the maximum degree is the network. As will be shown, if all destinations are stored locally the scheme uses \( \Theta(n \log(d)) \). A simple routing modification of the routing table data structure can improve the average storage requirements of a all-pairs routing algorithm (i.e. Dijkstra’s well known shortest path algorithm). Modify the routing table so that each entry corresponds to a channel \( c_t \) rather than a \( v \in V \). Each table entry \( t \) is associated with a list of intervals of form \([a, b]\), such that all destinations contained in \([a, b]\) use each channel \( c_t \). However, this modification does not improve the asymptotic space complexity of a destination based routing algorithm, since it is possible for many intervals to be associated with a single channel.

The two compact routing schemes that have received to most attention in research are interval routing and prefix routing. Interval routing (IR) was introduced by Santoro and Khatib [5]. IR is essentially the idea illustrated in the preceding paragraph, with the exception that the nodes are labeled in a ordered way that facilitates a low space complexity.

27
IR defines an interval as a set of consecutive integers. Each port is assigned an interval. A router finds the port used to forward a message by finding which port is labeled with the interval the destination id is contained within.

Interval Routing assumes that an Interval Labeling Scheme (ILS) is given. An ILS can be described as a 1-to-1 function \( F \) from \( v \in V \rightarrow n \in N \).

A Interval Routing Scheme (IRS) consists of two components. First, an input ILS. Secondly, for each processor \( i \), each \( \alpha \) in the set of ports \( P(i) \) of \( i \) are each labeled with an interval \((a, b)\) such that the set of intervals contained in \( P(i) \) is pairwise disjoint and covers \([1, n]\).

Several additional requirements on the definition of the interval routing scheme have proven useful. Fredrickson and Janardan studied the class of strict interval routing schemes [8]. A strict interval routing scheme is one where no interval label of processor \( i \) contains the node label of \( i \). Linear Interval Routing Schemes was introduced by Bakker, van Leeuwen, and Tan [9]. A linear interval routing scheme (LIRS) is a interval routing scheme where intervals are not allowed to 'wrap-around'. More formally, for all intervals \((a,b)\) of any processor \( i \), \( a \leq b \). Much research has been done in the area of linear interval routing. It has been shown that graph \( G \) has an optimal LIRS iff \( G \) is an outerplanar graph or \( G \) is \( K_4 \). [11]. It is intuitively clear that all graphs have an IRS, though the path may not be optimal, as shown in [5].

Prefix Routing was introduced by Bakker, von Leeuwen, and Tan [12]. They have given an optimal Prefix Routing algorithm for a tree and given some proofs on which graphs allow a optimal prefix routing scheme. Prefix Routing is closely related to Interval Routing. The idea is to label the node and ports with strings rather than integers as is used in interval routing. In their scheme, the port to use when forwarding a message is chosen by finding the longest port label that has the destination label as a prefix.
Section 3.2  Model And Notations

Each node is given a hardcoded, non-corruptible identifier which is assumed to be unique. It is assumed each node knows the identifiers of which nodes are its neighbors. Additionally, nodes are allowed to be labeled with a corruptible variable $label_i$ which is a logical address. Such a network is often referred to as a relabeled network.

The distributed system is modeled as an undirected, arbitrary graph $G=(V, E)$ consisting of a set of nodes $V$ and a set of edges $E$. $v \in V$ is an infinite state processor and $E$ is the set of links between $v_x$ and $v_y$. A link between two nodes indicates that they are neighbors. We use the notation $crash_{ij}$ to indicate link $(i,j)$ has failed and $crash_i$ to indicate processor $i$ has failed. It is assumed a self-stabilizing algorithm exists underlying the protocols that updates an input variable $Nbrs_i$. $Nbrs_i$ continually satisfies three conditions:

(i) \{\exists j : j \in Nbrs_i : (-crash_{ij}) \land (-crash_{ij})\},
(ii) i \notin Nbrs_i,
(iii) \{j \in Nbrs_i : i \in Nbrs_j\}.

We use the notation $Local\_Set_i$ to indicate $\{Nbrs_i \cup \{i\}\}$.

Each link may be assigned a label $l$ s.t. $l \in N$. We use $link_{ij}$ to indicate the label of edge $(i,j)$, and $L$ to indicate the set of all $\{i \in G, j \in Nbrs_i : link_{ij}\}$ at node $i$. This system is an asynchronous network of processors that communicate through a shared memory mechanism. The set of variables at processor $i$ is divided into two classes: field and local variables. Field variables may be read by a neighbor processor, but only written to by processor $i$. Local variables of processor $i$ may only be read and written to by $i$. The convention will adopted that variables are field variables unless specifically indicated as local.

A local state of processor $i$ is as a description of the variables and program counter of $i$. A global state is the set of all local states. A computation step is the atomic execution of a statement at a single processor. A computation is a sequence of computation steps.

The program at each node appears in the following form:

$< s_1 > \ldots < s_n >$. 
It is assumed each node has a program counter. Programs run in an infinite loop, i.e. execution of statement \( < s_n > \) sets the program counter to statement \( < s_1 > \). Statement are executed atomically. A node can read the states of its neighbors and write to its local memory in an atomic step. A scheduler exists, following the weak assumption of Distributed Demon.

The routing algorithms in this chapter make use of a pseudo code convention to show how messages are routed in the network. It is assumed that a lower layer of the network design handles issues of message verification and delivery. We use the following simple conventions for node \( i \in G \):

- **Send** \( (msg, dest, j) \) send message to nbr address \( j \) of \( i \) for destination address \( dest \).
- **Deliver** \( (msg) \) deliver message at node \( i \).
- **Receive** \( (msg, dest) \) receive message at node \( i \) for logical address \( dest \).

Several of the algorithms use that each connected component of the graph holds a spanning tree \( T = (V, E') \). This tree is built by a underlying self-stabilizing tree layer. Many self-stabilizing algorithms for spanning tree exist [17]. The following notations are used for \( T \):

- \( p(i) \) parent of node \( i \) in \( T \).
- **Root** id of the root of \( T \).
- \( \text{Depth}(T) \) height of \( T \), \( \text{Depth}(\text{Root}) = 0 \).
- \( \text{Depth}(i \in T) \) height of node \( i \) in \( T \).

### Section 3.3 Fault Model

A problem specification \( SP \) is a description of objectives. A legitimacy predicate \( LP \) for protocol \( P \) is a list of boolean predicates specified with respect to global state of \( P \), that when satisfied, invariantly satisfies \( SP \) in all computations of \( P \). Protocol \( P \) is said to be self-stabilizing if the following two conditions hold invariantly:

**Convergence:** Starting from an arbitrary global state, any computation of \( P \) reaches a global
state such that \( \mathcal{CP} \) is satisfied within a finite amount of time.

Closure: Any computation step taken while \( \mathcal{CP} \) is satisfied leaves \( \mathcal{CP} \) satisfied.

It is assumed there exists a certain point in time when faults will stop occurring for a sufficient finite period to allow stabilization the algorithms presented. The following exceptional events are listed below as a defining a unreliable topology.

**Permanent failure of links** A link between two nodes may simply fail-stop.

**Temporarily unavailable links** A link may become unavailable for a short rate due to maintenance purposes, excessive transmission error, or other malfunction.

**Transient Node Faults** A node’s state may become subject to internal perturbations that leave its state inconsistent with its program execution.

**Node Crash** A node may permanently become inactive with respect to other nodes.

**Links and Nodes joining the system** A new link may join the system and a previously failed link may become operational again. A new node or previously crashed node may be added to the system.

A fault model that tolerates only transient node faults, is defined as as a reliable topology.

An unreliable topology allows dynamic change of the network topology. Since dynamic additions of nodes and links are allowed, the topology can become arbitrary. As observed in Section 3.1, compact routing schemes are typically not well-defined on all topologies, which is problematic in constructing fault-tolerant compact routing schemes. However, observe that if we were to tenuously assume that no links are added to the system (most importantly, those that would increase the maximum degree of the network), the network topology would become limited to a subset of the original graph, which is a strong restriction. For example,
a graph $G$ that is a ring in the absence of any fault, may become a subgraph $G'$ of $G$ that is degree bounded by 2 in our fault model, if no links are added. This observation is followed a step further. It is pointed out here that the intent in this chapter is instead of assuming that no links can be added, it is assumed that maximum degree possible $D$ for a node in topology is known a priori. As a practical example, a ring topology may add nodes, add links and have nodes and links crash, but it is not expected to violate the degree bound of 2. It is clearly desirable that $D$ not be of an overly excessive value, since for most compact routing schemes, for example prefix routing, $D$ is closely related to the memory complexity of the protocol. If the bound of $D$ is illegally violated, it is assumed that a detection of this event will occur. The behavior in this situation of the compact routing algorithms in this paper is undefined.

Section 3.4 Compact Routing in Fundamental Topologies

Van Leeuwen and Tan [10] have shown that although for every graph there exists a IRS, it may not be the optimal scheme for that graph. Several popular topologies have optimal schemes, thus it is worthwhile to consider IR on these topologies. Santoro and Khatib gave proof of the existence of an optimal IRS for a tree and for a ring [5]. Additionally, optimal IR algorithms have been found for hypercube, torus, grid, and a variety of other topologies. Since there remains work to be done with fault-tolerant interval routing however, it is desirable to investigate this area.

One of the most simple topologies is a ring. Rings are a useful topology for control mechanisms (such as bandwidth arbitration). Compact routing optimally on a ring is much simpler to implement than on many other topologies, so it gives a good first example of a fault-tolerant compact routing algorithm. First a few preliminary definitions are necessary.
Section 3.4.1 Interval Routing Preliminaries

The IRS given in this chapter uses cyclic intervals. An cyclic interval \([a, b]\) is defined as:

\[
\begin{align*}
[a, a+1, \ldots, b-1] & \quad \text{if } a < b \\
[0, \ldots, b-1, a, \ldots, N-1] & \quad \text{if } a \geq b
\end{align*}
\]

Definition 3.2 Given a set of labels \(\mathcal{L}\), \(\mathcal{L} \uparrow k\) is defined as:

\[
\begin{align*}
\min(x \in \mathcal{L}) & \quad \text{if } k = \max(x \in \mathcal{L}) \\
\min(x \in \mathcal{L}|x > k) & \quad \text{if } k \neq \max(x \in \mathcal{L})
\end{align*}
\]

\(\mathcal{L} \uparrow k\) indicates the ‘next’ member in sorted, cyclic sequence of \(\mathcal{L}\) starting from \(k\).

Definition 3.3 An ILS is valid if each node has a unique label in \([0, n-1]\).

Definition 3.4 A path for an IRS is a sequence of adjacent nodes \(v_0, v_1, \ldots, d\) s.t for each \(v_i\), \(\text{label}_d\) in contained in some interval in \(\mathcal{L}(v_i)\).

Definition 3.5 An IRS is valid if:

1. \(\forall i \in G\), \(\mathcal{L}\) is pairwise disjoint and covers \([0, n-1]\).

2. \(\forall d, \forall s \in G\) there exists a unique path from \(s\) to \(d\).

Definition 3.5 An IRS is an optimal-path IRS if for each \(d \in G\), for each \(s \in G\), there exists a only a minimum-hop path from \(d\) to \(s\).

Section 3.4.2 Self-Stabilizing Compact Routing Algorithm for 2-degree Bounded Graphs

It is assumed the graph is bidirectional. Leeuwen and Tan [6] have shown that optimal ILS exists for a ring. However this algorithm is not fault-tolerant. Here is constructed a
fault tolerant algorithm with a modified, self-stabilizing version of the scheme of Leuwen and Tan. This scheme is designed for an unreliable topology, therefore $D = 2$ and it follows that once proven correct, the scheme works for any 2-degree bounded graph.

The optimal ILS for 2-degree bounded graph is a monotonic ILS. To create such a labeling in a self-stabilizing way intuitively requires a initial starting point upon which others nodes will follow. For this a spanning tree $\mathcal{T}$ is used, since many algorithms for leader election and spanning tree exist. $\mathcal{T}$ is used to elect the leaf node that has minimum id of any leaf node, hereto refereed as the leader. This node is the best choice to start a monotonic ILS for a connected component of $G$.

The labeling of the ports is achieved by a consistent orientation of the intervals. Note if the graph is a tree then the intervals for a node with two edges may be of drastically unequal sizes.

Section 3.4.2.1 Variables

$lid_i$: id of the leaf in subtree of $i$, or $NIL$ if $i$ is the root.

$ldr$: id of leaf with minimum id value.

Merely to create a simpler presentation of the algorithm, $n_s$ is used to indicate the number of nodes in the connected component that node $i$ belongs to. Clearly, computation of $n_s$ is trivial given $\mathcal{T}$. We also $c(i)$ to indicate the child of node $i$ in $\mathcal{T}$. $c(i)$ is undefined is $i$ is the root, and $NIL$ is $i$ is a leaf.

Section 3.4.2.2 Labeling Module

$S1: \{ \textit{Elect the leader} \}$

If ($i = \text{root}$)
\[ \text{lid}_i, \text{ldr}_i := \text{NIL}, \min(\{\text{lid}_j : j \in Nbrs_i : (p(j) = i) \} \cup \{\infty\}); \]

Else If (\(c(i) = \text{NIL}\))

\[ \text{lid}_i, \text{ldr}_i := i, \text{ldr}_{p(i)}; \]

Else

\[ \text{lid}_i, \text{ldr}_i := \text{lid}_{c(i)}, \text{ldr}_{p(i)}; \]

S2: { Label the nodes }

\(\text{If} (i = \text{ldr}_i)\)

\[ \text{label}_i := 0; \]

\(\text{Else if} (\text{ldr}_i \in Nbrs_i \text{ And } c(i) \neq \text{ldr}_i)\)

\[ \text{label}_i := n_s - 1; \]

\(\text{Else if} (\text{label}_i \neq \min(\text{label}_j : j \in Nbrs_i) + 1)\)

\[ \text{label}_i := \min(\text{label}_j : j \in Nbrs_i) + 1; \]

S3: { Label the edges }

\(\text{If} (\exists x \in Nbrs_i : \text{label}_x = \text{label}_i + n)\)

\[ \text{link}_{ix} := \text{label}_i + n - 1; \]

S4:

\(\text{If} (n_s = n \text{ And } \exists y \in Nbrs_i : \text{label}_y = \text{label}_i - n)\)

\[ \text{link}_{iy} := \text{label}_i + n_s \left\lceil \frac{n_s}{2} \right\rceil; \]

\(\text{Else if} (\exists y \in Nbrs_i : \text{label}_y = \text{label}_i - n)\)

\[ \text{link}_{iy} := 0; \]

Section 3.4.2.2 Routing Module

S1:

Receive (msg, dest)

Begin

\(\text{If} (\text{dest} = \text{label}_i)\)
Deliver (msg);
Else

Send (msg,dest) to j \in [dest, L \uparrow dest);
End

Section 3.4.2.2 Correctness Proof For 2-degree Bounded Algorithm

The proof of correctness use the convergence stair method. Convergence stair is an inductive procedure that shows that $L_n$ is self-stabilizing follows from $\{\forall n_0 : n_0 < n : L_{n_0} \text{ is self-stabilizing}\}$. A few additional notations for $G$ are introduced for the proofs.

$ML$ minimum leaf id in $T$.
$Leaf(i)$ id the leaf in the subtree of $i$ in $T$.
$G_s$ an arbitrary connected component of $G$.

A sense of direction must be defined to prove the correctness of IRS, even though the algorithm does not need an orientation. Consider the following definitions:

Definition 3.6 It is said that a path is monotonic if there exists a sequence of adjacent nodes $v_0, v_1, \ldots, v_x : label_{v_{i+n_1}} = label_{v_i} + n_1$. The direction of increasing labels is a monotonic increasing direction. A path in the opposite direction is called a monotonic decreasing direction.

The problem specification $SP$ is defined as : Given a 2-degree bounded graph, construct a valid, optimal-path IRS (see Section 3.4.1). The next step is to defined legitimacy predicate $L$ and show that when satisfied, it must satisfy $SP$. Refer to Section 3.2 if necessary.

$I_1 \equiv \{\forall i \in G : i \neq \text{Root} :: \text{lid}_i = \text{Leaf}(i)\}.$
$I_2 \equiv I_1 \wedge \{\forall i \in G : \text{lbr}_i = ML\}.$
$I_3 \equiv I_2 \wedge \text{Each } G_s \text{ forms a monotonic path}.$
$I \equiv I_3 \wedge \forall i \in G : \forall j \in Nbrs_i :: \forall k \in [\text{link}_{ij}, L \uparrow \text{link}_{ij}) \text{ } j \text{ is on an optimal path from } i \text{ to } k.$
Lemma 3.1 Eventually, $I_1$ converges and $ldr_{\text{Root}}$ equals $ML$.

Proof: The proof is done using induction on the height of $T$.

The base step is shown by observing that $S1$ is the only statement that modifies $lid_i$. Clearly if node $i$ is a leaf, then $lid_i = i$ follows from the rules. Next assume that

\[ \{ \forall k : 0 < k_0 < k < \text{depth}(T) :: ((i \in T) \land (\text{depth}(i) = k)) \Rightarrow (lid_i = \text{Leaf}(i)) \}. \]

It is claimed that \[ \{ \forall i \in T : \text{depth}(i) = k_0 :: (lid_i = \text{Leaf}(i)) \} \] follows. Node $i$ can only have one child in this instance, and chooses $lid_i$ of this child with depth equal to $k_0 + 1$. No other statements modify $lid_i$. All nodes of $T$ belong a subtree of the $\text{Root}$ by definition of $T$. The lemma follows from the induction hypothesis. \(\square\)

Lemma 3.2 $I_1$ is closed under system execution.

Proof: Assume that $I_1$ holds. Then for each subtree $T$ of the root, each node $i \in T$ has the same value for $lid_i$ and cannot assign a different value from its own by definition of $T$. \(\square\)

Lemma 3.3 Eventually, for each $i$ in $G$, $ldr_i = ML$. ($I_2$ converges)

Proof: The proof is done using induction on the height of $T$.

The base step follows from Lemma 3.1. Next assume that

\[ \{ \forall k : 0 < k_0 < k < \text{depth}(T) :: ((i \in G_1) \land (\text{depth}(i) = k)) \Rightarrow (ldr_i = ML) \}. \]

We claim that \[ \{ \forall i \in T : \text{depth}(i) = k_0 :: (lid_i = ML) \} \] follows. Node $i$ in this instance, chooses the $ldr$ of a parent $p$ with depth($p$) equal to $k_0 + 1$. No other statements modify $ldr_i$. The lemma follows from the induction hypothesis. \(\square\)

Lemma 3.4 $I_2$ is closed under system execution.

Proof: Similar to the proof of Lemma 3.2. \(\square\)
Definition 3.7 Let $LMP(s)$ be form the longest monotonic path $P$ of $G_s$ with the property that all labels in $P$ are less than $G_s \setminus P$. Let $LMax(s)$ be the node with the largest label in $LMP(s)$.

It will shown that $LMP(s)$ is a path that has in fact stabilized. The following step will be to show that eventually the length of $LMP(s)$ must continually increase to include all of $G_s$. Observe that we cannot choose simply the longest monotonic path because there may exist an adjacent node that destroys the monotonicity.

Lemma 3.5 If $k \in G_s$ and $k \notin LMP(s)$, then eventually $\text{label}_k > LMax(s)$.

Proof: Let $P = LMP(s)$. First observe that $\forall x \in P$ that $\text{label}_x$ is closed under system execution. $ML$ has only one assignment statement by $T_1$. $i \in P$ unequal to $ML$ cannot increase its value of $\text{label}_i$ by definition of $P$.

Let $U = G_i \setminus P$. If $U = \emptyset$ then of course we are done. Let $MVal = \min(\text{label}_x : x \in G_i)$. Let $M = \{x : \text{label}_x = MVal\}$. $v \in M$ must increase $MVal$, and thus in this case $MVal$ remains non-decreasing. Each $v \notin M$, cannot decrease the value $MVal$, and therefore in this case again $MVal$ remains non-decreasing. Each $v \in M$ must increase $MVal$, therefore $MVal$ remains increasing. It follows that $\{\forall x \in U, \forall y \in P : \text{label}_x > \text{label}_y\}$. \hfill $\square$

Lemma 3.7 Eventually, all nodes in each $G_s \in G$ converge to form a monotonic path ($T_3$).

Proof: The proof is on induction on the length of $LMP(s)$.

The base case is trivial, since only one assignment statement may execute for $ML$.

Assume that the length of $LMP(s)$ is $k_0 > 0$. Let $u$ be the node adjacent to $LMax(s)$ that is not in $LMP(s)$. Then $LMax(s)$ must be the smallest value in $LocalSet_u$ by Lemma 3.6. In fact, eventually by Lemma 3.6, all nodes not in $LMP(s)$ must have labels greater than $k_0$. Therefore, $u$'s conditional assignment to $\text{label}_u = \text{label}_{LMax(s)} + 1$ is true, and eventually $u$ is on a monotonic path to $ML$. The length of $LMP(s)$ is now at least $k_0 + 1$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Lemma 3.8 \( I_3 \) is closed under system execution.

Proof: Follows the fact that only \( ML \) or a node with a frond edge to \( ML \) can have an assignment statement execute infinitely. \( \square \)

Lemma 3.9 (\( \mathcal{L}_{RT} \) \( \Rightarrow \) (\( SP \) is satisfied.)

Proof:

It must be shown \( P \) for \( \mathcal{R} \) cover \([1,n] \) and are pairwise disjoint. The former clearly follows from the definition of \( \uparrow \) as a function defined over all of \([1,n] \). The proof of the latter follows from \( \uparrow \) defined as a many-to-one function.

That \( \mathcal{L}_{RT} \) holds indicates that a optimal, unique path between all nodes, satisfying the remainder of the definition of a valid, optimal IRS (see Section 3.4.1). \( \square \)

Lemma 3.10 \( I \) converges.

Proof:

case 1: \( G \) is a ring:

Assume by contradiction that an IRS path is longer than optimal. We have shown that \( I \) is pairwise disjoint. By the rules, the monotonic increasing direction must contain only labels of the form \([\text{label}_i + n \cdot 1, \text{label}_i + \lfloor \frac{n}{2} \rfloor] \) by definition of \( \uparrow \), the monotonic decreasing direction must contain only labels of the form \([\text{label}_i + \lceil \frac{n}{2} \rceil, \text{label}_i + n \cdot 1 \). Thus, it is obvious that packets can only proceed in one direction. Then it must use a port with an interval size greater than the diameter of the ring. Hence, a contradiction is obtained by inspection of the intervals at node \( i \) defined above.

case 2: \( G \) is not a ring:

Similar to case 1. \( \square \)

Lemma 3.11 \( I \) is closed under system execution.
Proof: Obvious from the rules.

Theorem 3.1 $R_I$ is self-stabilizing.

Section 3.4.4 Compact Routing Algorithm For Hypercubes

Another useful topology is a hypercube. Hypercubes have the advantage of both moderate degree and diameter and are commonly used in VLSI multi-processor design.

This section gives a fault-tolerant compact routing algorithm hereto referred to as $CH$ for a $d$ dimensional hypercube. $CH$ is self-stabilizing, routes using the optimal route, and uses constant amount of space after it has self-stabilized locally (which is easily checkable locally), assuming a reliable topology, refer to Section 3.3. Lan [14] has given a fault-tolerant algorithm for routing in hypercubes for an unreliable topology which which tolerates up to $d$ faults. His algorithm is not self-stabilizing, and does not always use optimal paths, however. It is observed that his algorithm does not label the nodes, instead assuming that they are hard-coded. In this section a self-stabilizing labeling algorithm for hypercubes is presented, and can easily be combined with Lan’s routing algorithm. $CH$ is primarily based on the ‘bit flipping’ algorithm. The basic idea is that each node of the hypercube differs from each of its neighbors by a single bit only. Nodes can route to each other using an exclusive or operation to decide the next node on a optimal path to the destination. The original source node creates a tag for the message by performing an exclusive or of its label and the destination label. An $i^{th}$ dimensional link is a link that connects to nodes that differ in the $i^{th}$ bit position only. A node $x$ receiving a message, sends by randomly choosing a bit position $k$ such that the message tag $t$ and $label_x$ differ in bit $k$. It sets bit position $k$ of tag $t$ to 0 and sends via the $k^{th}$ dimensional link. The fundamental idea of the execution of our algorithm is to utilize a leader denoted $ldr_1$ of the hypercube. Many self-stabilizing algorithms for leader election exist. The leader selects its own label to be 0. Each neighbor...
of the leader uses its id value in comparison with the \textit{Nbrs} of the leader to select a unique bit position to set to 1. All other nodes in the hypercube use an inclusive or operation of a subset its \textit{Nbrs} set to select their label. A difficulty in the algorithm is to prevent a faulty hypercube to continually performing an OR using faulty values. To overcome this problem, a local variable is maintained that is consistent with a breadth-first search tree that contains each node that has \( n \) bits in its label in a good state at level \( n \) in the BFS.

The following notations are used for bit sequences:

- \( x \downarrow y \) bitwise left SHIFT of \( x \), \( y \) positions to left.
- \( x \odot y \) \( x \) and \( y \) differ in exactly 1 bit position only.
- \( x \circ y \) sets bit position \( y \) of \( x \) to 0.

Section 3.4.4.1 Variables and Macros

\( dst_i \) variable describing distance to root.

\( ORDER_i(j) \) {
    returns the number of ids in \textit{Nbrs}\( j \) greater than \( i \);
}

\( \triangledown(i) \) {
    returns inclusive OR of \textit{Nbrs}\( i \) having \( d < d_i \), 0 if no such two exist.
}

Section 3.4.4.2 Labeling Module

\( S1: \{ \text{ remove cycles from graph } \} \)

\[
\text{If } (i = ldr_i)
\]

\[
dst_i := 0;
\]

\[
\text{Else}
\]

\[
dst_i := \min\{(d_j + 1 : j \in \textit{Nbrs}) \cup \{D\})\);
\]
S3: \{ Label the nodes \}

\[ i = ldr_i \]

\[ label_i := 0; \]

Else If \( ldr_i \in Nbrs_i \)

\[ label_i := 1 + (ORDER_i(ldr_i)); \]

Else

\[ label_i := \chi(i); \]

Section 3.4.4.3 Routing Module

S1:

Receive (msg, dest, tag)

Begin

If \( dest = label_i \)

Deliver (msg);

Else

Choose bit position \( k \) of \( tag \) equal to 1.

Let \( j \) equal the node on the \( k^{th} \) dimensional link from \( i \).

Send \( (msg, dest, tag \circ k, j) \)

End

Section 3.4.5 Correctness Proofs for Hypercube Algorithm

The problem specification \( S_P \) is defined as: Given a hypercube, construct a valid, optimal-path IRS (see Section 3.4.1). To prove correctness, again the convergence stair method is used. Legitimacy predicate is given \( L \) as follows:
\( I_1 \equiv \{ \forall i \in G, (i = \text{root} \land d_i = 0) \lor (i \neq \text{root} \land d_i = \min(d_j + 1 : j \in Nbrs_i) ) \} \)

\( I_2 \equiv \{ \forall i \in \text{LocalSet}_{ldr}, \forall j \in Nbrs_i : \text{label}_i \odot \text{label}_j \}. \)

\( I_3 \equiv I_2 \land \{ \forall i \in G, \forall j \in Nbrs_i : \text{label}_i \odot \text{label}_j \} \)

\( \mathcal{L}_{CH} \equiv I_3 \)

Lemma 3.11 \( I_1 \) is closed under system execution.

Proof: Only one assignment statement may execute for \( ldr \) hence is closed. Assume \( I_1 \) holds. Then for a node to assign a different value for its \( dst \) variable, it must have \( dst_i > D \). A contradiction is obtained by considering a chain of nodes from the \( ldr \) all which must have no rule enabled. □

Lemma 3.12 \( I_1 \) converges.

Proof:

The proof is similar to Lemma , using induction on the distance \( n \) from the \( ldr \).

Let \( P \) equal the set of nodes for which \( I_1 \) holds and have \( dst \) value no more than any node in \( M = G \setminus P \). Let \( P \) has clearly stabilized by definition of \( P \). Additionally observe that \( ldr \) trivially converges. It is straightforward to show that the smallest value in \( M \) continually increases past the highest value in \( P \) as done in Lemma 3.6. Therefore, by an inductive hypothesis on the number of hops from the leader, \( I_1 \) converges. □

Lemma 3.13 \( I_2 \) is self-stabilizing.

Proof: \( \forall k \in \text{LocalSet}_{ldr}, \text{label}_k \) can only be modified by one assignment statement, unconditionally executes, and depends only on \( id \) variables. Thus, \( I_2 \) converges and is closed. □

Lemma 3.14 \( I_3 \) is closed under system execution.

Proof: Assume that \( I_3 \) holds. \( \text{label}_{ldr} \) cannot change by definition of the rules. Pick arbitrary node \( x \neq ldr \). \( x \) chooses two values from its neighbor set for an inclusive or operation
given that they differ in two bits positions exactly. Therefore they must also in all other bit
positions agree with label\textsubscript{i} by definition of \( I_2 \) and are therefore closed. For the 2 bit posi-
tions that disagree, the only possible result since they disagree is two 1's of their inclusive
or. Therefore \( I_3 \) is closed.

Lemma 3.15 \( I_3 \) converges.

Proof: The proof is done using induction on the distance \( d \) from \( ldr \). We use the notation
\( LDst(x, d) \) to indicate node \( x \) is of minimum distance \( d \) from \( ldr \). Steps for \( d = 0 \) and \( d = 1 \)
are trivial.

Assume that \( I_3 \) holds for all \( v \in G \) with \( LDst(v, d) \) less than \( k \). It is claimed that event-
tually \( I_3 \) holds for \( v \) with \( LDst(v, d) \) equal to \( k \). By considering the BFS tree constructed,
i only chooses correct values from which to perform an inclusive OR upon. By definition
of the hypercube bit labeling and the induction hypothesis, the two smallest values or'd
always produce a label observing \( I_3 \).

The lemma follows from the induction hypothesis.

Theorem 3.2 \( CH \) is self-stabilizing.

Section 3.4.5 Self-Stabilizing Compact Routing Algorithm For Arbitrary Topologies

Interval routing is an effective routing scheme for simple cyclic graphs like a ring. Espe-
sially since it is has been shown in [12] that no optimal prefix algorithm exists for a graph
containing a cycle of more than 4. But IR is well-known to have a number of disadvantages
when applied to arbitrary topologies. Fault tolerance issues are not always well-addressed
by interval routing schemes. For example, a depth-first labeling scheme easily requires all
nodes reassign labels in response to a single node or link failure.

Prefix Routing (PLS) was introduced by Bakker, von Leeuwen, and Tan [12] as an
Improvement on the deficiencies of Interval Routing. A popular data structure used in interconnection network implementations, called a trie, utilizes exactly this idea for quick routing table lookup. The idea is to label the node and ports with strings. For each parent, each child picks a unique symbol from an input alphabet \( \Sigma \) and extends its parent's label by that symbol. The port label to a route a message to is the longest label containing a prefix of the message label. Essentially, the improvement of PLS over ILS comes from using \textit{breath-first} traversal rather than depth-first. In this way, only the subtree of \( i \) needs to recompute when \( i \) or link \((i, p(i))\) fails.

Section 3.5.2.1 Variables and Macros

\( x \cdot y \) to indicates the string obtained by concatenating \( y \) onto the end of \( x \). \( \epsilon \) indicates the \( \textit{NIL} \) string. \( \Sigma \) is a set of symbols, containing \( D \) unique elements. The notation \( \Sigma[i] \) indicates the \( i^{th} \) element of \( \Sigma \). \( |x| \) indicates the size of string \( x \).

- \( PFX(x, y) \) indicates: if \( x \) is not a prefix of \( y \) returns \( \epsilon \), else returns the string obtained by removing the maximal common prefix of \( x \) and \( y \) from \( x \).

- \( FROND SEARCH_i(dest) \) is a function returning the result of a binary search of all frond edges of \( i \) for a frond edge \( \alpha \) containing the longest string \( \delta \) s.t. \( \delta \) is a prefix of \( dest \). If no frond edge exists that contains a prefix of \( dest \), \( FROND SEARCH_i(dest) \) returns \( \epsilon \).

- \( FIRSTSYM_i(d) \) returns the \( i \) such that first symbol of string \( d \) is equal to \( \Sigma[i] \).

- \( d_{frond} \) indicates a local variable of node \( i \), of type integer.
Section 3.5.2.2  Labeling Module

This section gives a self-stabilizing version of the algorithm of Bakker, von Leeuwen, and Tan.

S1: \{ label the nodes \}

If (i = root)

\[ label_i := e; \]

Else

\[ label_i := \text{label}_p(i) \cdot \sum [ORDER_p(p(i))]; \]

S2: \{ label the nodes \}

For (\forall j \in Nbrs_i)

\[ \text{link}_{ij} := \text{label}_j; \]

Section 3.5.2.3  Routing Module

The routing algorithm is slightly optimized for quicker routing. By an abuse of notation, link variables are used as an array such that all tree edges are stored in the first part of the array. Then is it possible to first binary search the frond edges only. If no suitable frond edge is found, then the algorithm indexes into the link array to find the appropriate tree edge to use.

Note the correctness reasoning of any prefix routing algorithm follows from continually reaching nodes that contain a longer prefix of the destination.

S1:

Receive (msg, dest)

Begin

If (dest = label_i)

Deliver (msg);
Else

\[ \text{dfrond}_i := \text{FROND.SEARCH}_i(\text{dest}); \]

If (|PFX(\text{dest}, \text{label}_i)| \geq |PFX(\text{dfrond}_i, \text{label}_i)|)

If (|\text{label}_i| \geq |\text{dest}|)

Send (msg, \text{dest}) to \text{label}_{p(i)}; 

Else

Send (msg, \text{dest}) to \text{link}_i[\text{FIRSTSYM}(\text{tstring})];

Else

Send (msg, \text{dest}) to dfrond_i;

End

Section 3.5.3 Proofs of Correctness for PR Algorithm

The problem specification is defined as: Given an arbitrary graph, construct a valid PLS. The convergence stair method is used as usual. Legitimacy predicate \( \mathcal{L} \) as such:

\[ \mathcal{I}_1 \equiv \text{each child of a parent has a unique label} \]
\[ \mathcal{I}_2 \equiv \mathcal{I}_1 \land \text{each child extends its parent by one symbol.} \]
\[ \mathcal{I}_3 \equiv \mathcal{I}_2 \land \text{each packet eventually reaches its destination.} \]
\[ \mathcal{L} \equiv \mathcal{I}_3 \]

It should be sufficient to state that \( \mathcal{I}_1 \) and \( \mathcal{I}_2 \) are self-stabilizing by an inductive hypothesis. With this in mind, \( \mathcal{L} \) is proven by only proving one lemma.

Lemma 3.16 Each packet using PR eventually reaches its destination.

Proof:

Three possible cases are consider for node \( s \) sending to node \( d \).

case 1: \( d \) is an ancestor of \( s \)

No frond edge of node \( s \) can contain a prefix of node \( d \). Therefore \( \text{dfrond}_i \) will equal \( \epsilon \),
and $s$ sends using a tree edge. Since all children of $s$ extend the label of $s$, $s$ sends to its parent.

**case 2:** $d$ is a descendant of $s$

Similar to case 1.

**case 3:** $d$ is neither an ancestor or descendant of $s$

If $s$ does not have a frond edge leading to a node $z$ which has $d$ in its subtree then $s$ will send to the root. Thus in this instance, case 3 reduces to case 2. If $s$ does have a frond edge to such a node $z$ then $s$ sends via this edge and again case 3 reduces to case 2. □
BIBLIOGRAPHY


[36] Ming Shin Tsai and Shing Tsaan Huang, "A Self-Stabilizing Algorithm for the Shortest Paths Problem with a Fully Distributed Demon".