A self-stabilizing interval routing scheme in general networks

Doina Bein
University of Nevada, Las Vegas
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.

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.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A SELF-STABILIZING INTERVAL ROUTING SCHEME IN GENERAL NETWORKS

by

Doina Bein

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

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

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2001
The Thesis prepared by

Doitia Bein

Entitled

A Self-Stabilizing Internal Routing Scheme in General Networks

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

Master of Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Self-Stabilizing Interval Routing Scheme in General Networks

by

Doina Bein

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

Routing is the term used to describe the decision procedure by which a node selects one (or, sometimes, more) of its neighbors to forward a message on its way toward the final destination. The amount of information kept in each node for routing must be as small as possible, but a message should be delivered on as short a path as possible. The Interval Routing Scheme (IRS) labels the nodes with unique integers from a contiguous range and labels the outgoing arcs in every node with a set of intervals forming a partition. Since IRS has an impact on routing in the global Internet, the design and implementation of a distributed, fault-tolerant, and robust IRS is an important research topic. The Pivot Interval Routing (PIR) scheme [EGP98] divides the nodes in the network into pivots and clients of the pivots. A pivot acts as a center for the partition of the network formed by its clients. Each node can send messages directly only to a small subset of vertices in its nearby vicinity or to the pivots.

An algorithm is called self-stabilizing [Dij74] if, starting from an arbitrary initial state, it is guaranteed to reach a correct state in finite time and with no exterior help. In this thesis, we present a self-stabilizing PIR algorithm. The algorithm starts with no knowledge of the network architecture and, eventually, each node builds its own routing table of size $O(n^{1/2} \log^{3/2} n + \Delta_v \log n)$ bits ($n$ is the number of nodes in the network and $\Delta_v$ is the degree...
of node \( v \), with a total of \( O(n^{3/2} \log^{3/2} n) \) bits. The stretch factor (the ratio between the length of a path and the distance between the endpoints) is at most five and is three on the average. The stabilization time of the algorithm is \( O(d \sqrt{n(1 + \log n)}) \) time units, where \( n \) is the number of nodes and \( d \) is the diameter of the network.
# TABLE OF CONTENTS

ABSTRACT .................................................. iii

LIST OF FIGURES ....................................... vi

ACKNOWLEDGMENTS ...................................... vii

INTRODUCTION ........................................ 1
  Related Work ........................................... 2
  Contributions .......................................... 3
  Outline of the Thesis ................................. 4

PRELIMINARIES .......................................... 5
  Distributed Systems .................................... 5
  Programming Notations ............................... 7
  Self-Stabilization ..................................... 8

PIVOT INTERVAL ROUTING SCHEME ............. 11
  Routing Schemes ....................................... 11
  Interval Routing Scheme ............................ 14
  Pivot Interval Routing Scheme ..................... 16
    Overview of PIR ....................................... 17

THE DISTRIBUTED PIVOT INTERVAL ROUTING SCHEME .... 22
  Data Structures ....................................... 24
  The Algorithm ......................................... 26
    Error_Correction .................................... 28
    Calculate_Ball ....................................... 29
    Partial Algorithms - Second phase .............. 33
    To_Clients and To_Pivots ......................... 35
    Pivot_Label and Client_Label .................... 40
  One example ........................................... 45

PROOF OF CORRECTNESS ............................. 47
  A Correct T-ball ....................................... 48
  Choosing the Nearest Pivot ....................... 57
  Relabeling ............................................. 59
  Self-Stabilization and Time Complexity.......... 62

CONCLUSIONS .......................................... 66

BIBLIOGRAPHY ......................................... 67
LIST OF FIGURES

3.1 An arbitrary network ................................................................. 18
3.2 The t-ball $B_1(5)$ ................................................................. 19
3.3 The pivots and their clients ...................................................... 21

4.1 A partial t-ball for node 1 ...................................................... 45
4.2 The t-ball $B_1(5)$ ................................................................. 46

5.1 A correct distance replaces the old one ................................. 49
5.2 The node $u$ reached $v$ through $w$ but the path $u \rightarrow w$ is disconnected ................................. 50
5.3 A crash in the direct neighborhood ....................................... 51
5.4 The down node $u$ reached $v$ through $w$ ............................. 51
5.5 A cycle in delivering of dimension 3 ....................................... 52
5.6 A correct situation ................................................................. 52
5.7 A shorter distance through another neighbor ....................... 55
5.8 Increasing min. distance ........................................................ 59
ACKNOWLEDGMENTS

I would like to thank Dr. Ajoy Datta for chairing my committee and advising this work. For this and for being generous with his time when I needed it I am deeply indebted to him. I would also like to specifically thank Dr. John Minor, Dr. Tom Nartker, and Dr. Henry Selvaraj for serving on the committee. I would like to thank the faculty of the Computer Science Department of the University of Nevada, Las Vegas for the formal education along with generous financial support. I am grateful to the Graduate College for having supported my work through a summer stipend during the final stages of this work.

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

INTRODUCTION

Routing schemes implemented in point-to-point communication networks deliver messages between processors. For large networks it becomes important to reduce the amount of memory kept in each node for routing. At the same time, a message should be delivered on as short a path as possible. For literature on the routing protocols, see [Tel94] and [Gou98].

A universal routing strategy is an algorithm which generates a routing scheme for any given network. A compact routing scheme uses compact routing tables. A table contains an entry, for each link of a node, specifying which destinations must be routed via this link [Tel94]. The most popular such scheme is the interval routing scheme IRS ([SK85]), which labels the nodes with unique integers from a contiguous range, and labels the outgoing arcs in every node with a set of intervals forming a partition of the name range. A message invoking the delivery protocol is sent on the unique outgoing arc labeled by an interval which contains the destination label. While the preprocessing step is complex, the delivery protocol consists of simple “shoot and forget” decision functions in every node, which depend only on the destination.

It is important to update the routing scheme dynamically in case of network change (processors can be added or conceivably crash.) The most general technique of designing a system to tolerate arbitrary transient faults is self-stabilization ([Dij74].) A self-stabilizing system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the processors and initial messages on the links. In a distributed self-stabilizing algorithm, a node, with no initialization code and having only local information, has to achieve a global objective, to build a correct IRS in the network.
1.1 Related Work

Among all the research done in the area of compact routing, the hierarchical routing scheme is the most related to our work. It is known [ABNL-89, ABNLP90, AP90, PU89] that the memory requirements of a routing scheme are related to the worst case stretch factor the routing scheme guarantees. Peleg [PU89] showed that any universal routing strategy which can achieve a stretch factor $s \geq 1$ must use a total of $\Omega(n^{1-\frac{1}{s+1}})$ bits of routing information in the network.

Several routing strategies have been proposed which achieve an almost optimal efficiency-space relation. Specifically, Peleg [PU89] proved that for every graph and every integer $k \geq 1$ it is possible to construct a hierarchical routing scheme with stretch factor $O(k)$ which uses a total of $\Omega(k^3n^{1+1/k}\log n)$ bits and labels each node with $O(\log^2 n)$ bits. The scheme has a few drawbacks: it is not name-independent (it relabels the nodes with new names), it does not bound the local memory requirement of a node, and finally, it assumes a unit cost on the links of the network. Other hierarchical routing methods, see [ABNL-89] and [AP90], avoid these problems but at the price of non-optimal efficiency-space. But the major disadvantage of all proposed hierarchical routing strategies is a complex decision function at the nodes, which becomes a bottleneck in the case of high-speed networks.

The first compact routing method was proposed by Santoro and Khatib [SK85]. The method is based on labeling the nodes of a tree with integers from 0 to $N - 1$ ($N$ represents the total number of nodes in the tree) in such a way that the set of destinations for each link is an interval. Van Leeuwen and Tan [LT87] extended the tree labeling scheme to non-tree networks in such a way that (almost) every link is used for packet traffic. Eilam [EGP98] presented a particular kind of interval routing scheme called pivot interval routing scheme (PIR.) Each node has a routing table of size $O(n^{3/2}\log^{3/2} n)$ bits and knows only its direct neighbors. The scheme is not name-independent, but it bounds the local memory requirement of a node, and makes no assumptions of the costs of the links in the network.

In 1973, Dijkstra introduced the notion of self-stabilization in the context of distributed systems [Dij74, Dij82]. He defined a system as self-stabilizing when, regardless of its initial
state. It is guaranteed to arrive at a legitimate state in a finite number of steps. A system which is not self-stabilizing may stay in an illegitimate state forever.

Fault-tolerance is an important issue in designing network routing protocols since the topology changes due to the link/node failure or recovery. An optimal self-stabilizing shortest path tree construction is presented in [AKM+93]. Self-stabilizing topology-update problems are discussed in [APSV94, Dol97, DH97, GS95, Mas95].

1.2 Contributions

Eilam [EGP98] presented the $\text{VXTZ}$ scheme, which assumes a full knowledge of the network for all the nodes in the graph. However, they gave no algorithm for its implementation. The article mentioned that such a scheme can be constructed in time polynomial in $N$, the number of the nodes. $\text{VXTZ}$ is similar to a 2-level hierarchical routing scheme, in the sense that it involves 2-level message passing in case the source and the destination are not in the same neighborhood.

In this thesis we propose a self-stabilizing pivot interval routing strategy $\text{SPIR}$. It is a fault-tolerant distributed implementation of the $\text{PIR}$ scheme in a general asynchronous network. The algorithm starts with no knowledge of the network architecture and progressively builds an IRS scheme. It supports fault causing nodes and link failures and additions of nodes and/or links, and it is guaranteed that it will reach a correct state in finite time (it is self-stabilizing.)

The tasks are fairly distributed among the nodes and each node builds its own routing table based on the information gathered online up to the current moment. There are nodes in the network (called pivots) which play the role of the routers. Their routing table capacities are the same as the non-pivots nodes $O(n^{1/2} \log^{3/2} n + \Delta_v \log n)$ bits, where $n$ is the number of nodes in the network and $\Delta_v$ is the degree of node $v$. with a total of $O(n^{3/2} \log^{3/2} n)$ bits. The stretch factor of five and the average stretch of three are preserved, as in the $\text{PIR}$ scheme. The stabilization time of the algorithm is $O(d \sqrt{n(1 + \log n)})$ time units, where $n$ is the number of nodes and $d$ is the diameter of the network.
1.3 Outline of the Thesis

In the next Chapter we give general definitions for distributed systems and self stabilization. Since the PIR scheme is a particular case of IRS, the interval routing scheme and some specific definitions are given in Chapter 3. Chapter 4 contains our main contribution, the distributed PIR algorithm. We then prove the correctness of the algorithm in Chapter 5 and we give some concluding remarks in Chapter 6.
CHAPTER 2

PRELIMINARIES

In this chapter we present a number definitions for a distributed system and self-stabilization, and we give precise programming notations used in the algorithm.

2.1 Distributed Systems

A recent orientation in computer systems research is to consider distributing computation among several processors (multiprocessor systems.) If the processors share the computer bus, the clock, and sometimes memory and peripheral devices, the systems is called tightly coupled. If the processors do not share memory or clock, and instead, have their own memory, they are called distributed systems. The processors communicate with each other through various communication lines, such as high-speed buses or telephone lines.

Definition 2.1.1 (Distributed System) A distributed system is an undirected connected graph, $S = (V, E)$, where $V$ is a set of nodes ($|V| = n$) and $E$ is the set of edges. Nodes represent processors, and edges represent bidirectional communication links.

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

There are many instances in which programs need to exchange data among each other on the system. When the processors are connected by a communication network, the processes
at different sites can exchange information. There are two common models of communication: message passing and shared memory. In the message passing model, information is exchanged through an interprocess-communication facility (IPC) provided by the operating system. In the shared memory model, processes use map memory system calls to gain access to regions of memory owned by other processes. A process can access a region of memory owned by another process. Processes may exchange information by reading and writing data in these shared areas. The form of the data and the location are determined by these processes and are not under the operating system’s control.

IPC offers a mechanism to allow processes to communicate and synchronize their actions, providing at least two operations: send(message) and receive(message). If processes $p_i$ and $p_j$ need to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. Messages sent by a processor can be either fixed size or variable size.

Regarding the timing of events (receiving/delivering a message, computing local information), we have several synchronous, asynchronous, and partially synchronous models. The synchronous model is the simplest model to describe and to program. We assume that all processors take steps in their executions simultaneously, and the transmission time of each message is bounded. But this is very difficult to implement, and because of this, most distributed systems are not synchronous. The asynchronous model is the other extreme, where processors can take steps at arbitrary speeds and in arbitrary orders. It is the hardest to program, because of the uncertainty in the order of events. Since the asynchronous model has no assumption about time, algorithms designed for the asynchronous model are general and portable: they are guaranteed to run correctly in networks with arbitrary timing guarantees. On the other hand, the asynchronous model does not provide sufficient conditions to solve problems efficiently, or even to solve them at all. The partially synchronous model is in between, with a wide range of possible assumptions that can be made. A very common assumption is to bound the interval of time for transmitting a message, called timeout, after which the message is considered lost.
We consider networks which are asynchronous. Each node starts with an unique ID chosen from some large totally ordered space of identifiers such as the positive integers, \( \mathbb{N}^+ \). All process ID's in the network are distinct, but there is no constraint on which IDs actually appear in the network (they do not have to be consecutive integers.) Initially, each node knows only its direct neighbors. Edges are labeled by distance values. A communication link \((p, q)\) exists if and only if \(p\) and \(q\) are neighbors and the physical distance between them identifies the length of the link.

Every processor \(p\) can distinguish all its links. The variable \(N_p\) refers to the set of the direct neighbors of \(p\), arranged in some arbitrary order \(<_p\). The number of neighbors of \(p\), \(|N_p|\), is called the degree of \(p\) and is denoted by \(\Delta_p\). We assume that \(N_p\) is maintained by an underlying local topology maintenance protocol that it can alter its values in case of changes in the network (failures of processors, or links, or both.)

2.2 Programming Notations

Each component of a system (node or link) has a local state, which is defined by the ID of the node and the values of the program variables. We define the global state of a system as the union of the local state of its components.

The program consists of a set of global variables and a finite set of actions. A process can read/write its own variables and only read the variables of the neighboring processors.

Each action is uniquely identified by a label and is part of a guarded command:

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

The guard of an action is a Boolean expression involving the global variables and/or local variables. The action can be executed only if its guard evaluates to \(true\). We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding action, if it is selected for execution, are done in one atomic step.

In the system, one or more processors execute an action and a processor may take at most one action. This execution model is known as the distributed daemon. We assume a weakly fair daemon, meaning that if processor \(p\) is continuously enabled, \(p\) will be eventually
chosen by the distributed daemon to execute an action.

A network protocol is a set of node programs, one for each node. A global state of the protocol is the state of all nodes as well the messages on links.

Let $C$ be the set of all possible configurations of the system and a distributed protocol $P$ be a collection of binary transition relations denoted by $\rightarrow$ on $C$.

A computation of $P$ is a maximal sequence of configurations $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \ldots)$ such that $\forall i \geq 0$; $\gamma_i \rightarrow \gamma_{i+1}$ if $\gamma_{i+1}$ exists or $\gamma_i$ is a terminal configuration. Maximal means that the sequence is either infinite, or it is finite and no action of $P$ is enabled in the final configuration.

The set of all possible computations of $P$ in the system $S$ is denoted by $E$.

The set of computations of $P$ in $S$ starting with a particular configuration $\alpha \in C$ is denoted by $E_\alpha$. A configuration $\beta$ is reachable from $\alpha$, $\alpha \sim \beta$, if there exists a computation $e \in E_\alpha, e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i-1}, \ldots)$ such that $\beta = \gamma_i, i \geq 0$.

2.3 Self-Stabilization

The notion of self-stabilization was introduced to computer science by Dijkstra in 1973. He limited his attention to a ring of finite-state machines. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps" [Dij82]. A non self-stabilizing system may stay in a non legitimate state forever. Dijkstra observed that "local actions taken on account of local information must accomplish a global objective" [Dij82].

His work was rather incomprehensible, and it was Lamport in 1983 who appreciated and explained it. Since then, substantial research was done and particular cases of self-stabilization have been studied in articles: snap SS, fault containing, super-stabilization [Do100]. The work of Dijkstra remains a "milestone in work of fault tolerance" and is now considered to be the most general technique for designing a system to tolerate arbitrary transient faults.

A self-stabilizing system $S$ guarantees that, starting from an arbitrary global state, it reaches a legal global state within a finite number of state transitions, and remains in a legal
state unless a change occurs. In a non-self-stabilizing system, the system designer needs to 
enumerate the accepted kinds of faults, such as node/link failures, and he must add special 
mechanisms for recovery. Generally, not all types of faults are taken in consideration, and 
an obscure error such as a memory corruption can provoke a general reset of the entire 
system. Ideally, a system should continue its work by correctly restoring the system state 
whenever a fault occurs ([AG93, Gou98].)

Self-stabilization offers a uniform mechanism to cope with not only arbitrary transient 
faults such as data, message, location counter corruption ([KP93]), but also with a variety 
of faults such as network congestion and software bugs ([LAJ99].)

Given a predicate \( P \): among \( C \), the set of all possible configurations, we denote by \( \mathcal{L}_P \) 
the set of legitimate configurations, (corresponding to the legitimate states of the system, 
which satisfy \( P \)), and by \( C - \mathcal{L} \) the set of illegitimate configurations (which do not satisfy 
\( P \)). The predicate truth of the \( P \) for our distributed algorithm guarantees that the network 
has a correct \( \mathcal{PRI} \) scheme.

The relation \( c \vdash P \) means that an element \( c \in C \) satisfies the predicate \( P \) defined on 
the set \( C \). A predicate is non-empty if there exists at least one element that satisfies the 
predicate. We define a special predicate \( true \) as follows: \( \text{for any } c \in C, c \vdash true \).

We introduce the concept of an attractor to define self-stabilization. Intuitively, an at­
ttractor is a set of configurations of the system \( S \) that "attracts" another set of configurations 
of \( S \) for any computation in \( \mathcal{E} \).

**Definition 2.3.1 (Closed Attractor)** Let \( C_1 \) and \( C_2 \) be subsets of \( C \). \( C_1 \) is an attractor 
for \( C_2 \) if and only if for any initial configuration \( c_1 \) in \( C_2 \), for any execution \( e \) in \( \mathcal{E}_{c_1} \), 
\((e = c_1, c_2, \ldots). there exists i \geq 1 such that for any j \geq i, c_j \in C_1.\)

We can define the closed attractor in terms of predicates:

If \( C_1 \) and \( C_2 \) are the set of configurations satisfying predicate \( P_1 \), and respectively \( P_2 \), then 
\( P_2 \) is an attractor for \( P_1 \) if and only if \( \forall \alpha \vdash P_1 : \forall e \in \mathcal{E}_\alpha : e = (\gamma_0, \gamma_1, \ldots) : \exists i \geq 0, \forall j \geq 
i, \gamma_j \vdash P_2 \). We denote this relation as \( P_1 \triangleright P_2 \).

In the usual (non-stabilizing) distributed system, the possible computations can be
restricted by allowing the system to start only from some well-defined initial configurations. On the other hand, in a stabilizing system, problems cannot be solved using this convention, since all possible system configurations are admissible initial configurations.

**Definition 2.3.2 (Self-stabilization)** A system $S$ is called self-stabilizing if and only if there exists a non-empty subset $\mathcal{L} \subseteq \mathcal{C}$ of legitimate configurations and a predicate $\mathcal{L}_p$ such that $\mathcal{L}$ is a closed attractor for $\mathcal{C}$:

(i) $\forall \alpha \vdash \mathcal{L}_p : \forall e \in \mathcal{E}_\alpha :: e \vdash \mathcal{S} \mathcal{P}_p$ (correctness).

(ii) $true \triangleright \mathcal{L}_p$ (convergence).
CHAPTER 3

PIVOT INTERVAL ROUTING SCHEME

We start this chapter by giving an overview of the routing scheme in general. Then we continue with interval routing schemes and we present a particular case of IRS called pivot interval routing scheme.

3.1 Routing Schemes

Generally, networks are not fully connected. Moreover, in case of super-networks (networks of networks), we often have sparse networks. So, for nodes to communicate with each other in a connected network, we have to define a procedure such that somehow a decision is taken and messages will reach their destination. A node can communicate directly (to send packets of information) only to a subset of the nodes called the neighbors of the node. Routing is the term used to describe the decision procedure by which a node selects one (or sometimes many) of its neighbors to forward the message on its way toward the final destination. The objective in designing a routing strategy is to generate, for every node, a decision-making procedure to perform this function and guarantee delivery of each message. In the extreme case, we can have a simple procedure: send the message to all neighbors, each neighbor does the same (flood the network with messages.) Of course the message will reach eventually the destination, but the network is overloaded with too many messages. In this case we do not need to know anything about the network, so no memory is required locally for the decision function. If we want to avoid this situation we can proceed to analyze the network and gather information about its topology. The more information we have, the better is the decision function. This means that the message will follow a shorter path to
its final destination.

Assume that some process $p_i$ (abstracted by node $u$) wants to send a message to process $p_j$ (abstracted by node $v$.) This will be done along some route, which is a sequence of adjacent communication links in the network (abstracted as a simple path.) A routing algorithm specifies the route by telling each intermediate node on the route on which outgoing edge the message should be sent depending on the destination.

The information stored in each node regarding the network topology as a working basis for the local decision procedure is called routing table and defines the memory required by that node. The total memory required by the strategy is the sum of the memory required in each node of the network and it is a performance criterion of the algorithm. The routing problem has two parts:

1. **Preprocessing** - the routing table computation. The routing table must be computed when the network is initialized or must be brought up-to-date when a change occurs.

2. **Delivery** - when a message is to be sent through the network, it must be forwarded using the local routing table.

Also, in analyzing a routing scheme we must take into consideration some factors [Tel94]:

1. **Correctness**: the algorithm must deliver every message sent through the network to its final destination.

2. **Communication complexity**: the algorithm for the computation of the routing tables must use as few messages, time and storage as possible.

3. **Efficiency**: the algorithm must send messages through paths that are as good as possible. The term of good refers to the minimization of the delay and high throughput of the entire network. An algorithm is called optimal if it uses always the "best" paths.

4. **Robustness**: in case of a topology change (addition/removal of a channel or node,) the algorithm updates the routing tables in order to perform the routing function in the modified network.
5. **Adaptativeness**: the algorithm balances the load of channels and nodes by adapting the tables in order to avoid paths through channels or nodes that are highly busy, preferring channels and nodes with a currently light load.

6. **Fairness**: the algorithm must provide service to every node in the same degree.

The term of "best" path can mean different things:

- **Shortest path**: each link is characterized by a *length* value, and the cost of the path represents the sum of the length of the links in the path.

- **Minimum hop**: a path is measured by the number of hops (traversed links or steps from one node to the other.)

- **Minimum delay**: each link has dynamically assigned a *weight*, depending on the traffic on the channel.

An algorithm would concentrate on one of these aspects and try to choose the path among all the paths that link two nodes, which has the minimum value.

Therefore, each processor \( p_i \) maintains a routing table which contains, for each destination \( p_j \) in the network, \( p_j \neq p_i \), the identity of \( p_i \)'s neighbor on the path to \( p_j \). In order to always have the shortest path between any two nodes, the total memory required for each node is \( O(n \log d) \) bits (therefore \( O(n^2 \log n) \) for the entire network.) The distributed algorithm to solve this case is easier.

The three most common routing schemes are *fixed routing*, *virtual routing*, and *dynamic routing*:

- **Fixed routing**: a path from \( p_i \) to \( p_j \) is specified in advance and does not change unless a hardware failure disables this path. Usually, the shortest path is chosen, so the communication costs are minimized.

- **Virtual routing**: a path from \( p_i \) to \( p_j \) is fixed for the duration of one *session*. Different sessions involving messages from \( p_i \) to \( p_j \) may have different paths.
• Dynamic routing: the path used to send a message from \( p_i \) to \( p_j \) is chosen only when a message is sent. Because the decision is made dynamically, separate messages may be assigned different paths.

There are tradeoffs among these three schemes. Fixed routing cannot adapt to link failures. If a path has been established between \( p_i \) and \( p_j \), the messages must be sent along this path, even if the path is down. We can partially remedy this problem by using virtual routing, and can avoid it completely by using dynamic routing. Fixed and virtual routing ensure that messages from \( p_i \) to \( p_j \) will be delivered in the order in which they were sent. In dynamic routing, messages may arrive out of order.

The dynamic routing is the most complicated to set up and run, but is the best way to manage routing in complicated environments.

Some routing strategies code topological information in the address of a node, in order to use shorter routing tables or fewer table lookups. These so-called “compact” routing schemes do not always use optimal paths.

3.2 Interval Routing Scheme

An interval routing scheme is a way of implementing routing schemes on arbitrary networks. It is based on representing the routing table stored at each node in a compact manner by grouping the set of destination addresses that use the same output port into intervals of consecutive addresses (nodes of the graph representing the network.)

As originally introduced in [SK85], where the scheme required each set of destinations to consist of a single interval, it has been subsequently generalized in [LT87] to allow more than one interval per edge.

Consider an undirected \( n \)-node graph \( G = (V, E) \). Since \( G \) is undirected, each edge \((u, v) \in E\) can be viewed as two arcs, i.e. two ordered pairs, \((u, v) \) and \((v, u)\).

**Definition 3.2.1 (Interval Routing Scheme)** An interval routing scheme \( R \) on \( S \) is a routing scheme consisting of a pair \((\mathcal{L}, I)\), generated in the preprocessing step, where \( \mathcal{L} \) is a node-labeling, \( \mathcal{L} : V \rightarrow \{1, \ldots, n\} \), and \( I \) is an arc-labeling, \( I : E \rightarrow 2^{\mathcal{L}(V)} \). Formally, for
every \( x \in V \), the collection of sets that label all the outgoing arcs of \( x \) forms a partition of the name range (possibly excluding \( x \) itself):

1. \( \bigcup_{e \in E_x} I(e) \cup L(x) = \{1, \ldots, n\} \)

2. \( I(e) \cap I(e') = \emptyset \), for every two distinct arcs \( e, e' \) in \( E_x \)

The delivery protocol is defined as follows: the message is sent on the arc \( e \) labeled by a set \( I(e) \) that contains the destination (destination \( \in I(e) \)).

The graph \( G \) is said to support an interval routing scheme (IRS) if there exists this pair \( \mathcal{R} = (\mathcal{L}, \mathcal{I}) \).

An IRS can be characterized by two notions: compactness and stretch factor.

The compactness of an arc \( e \in E \), \( c(I(e)) \), represents the minimum number of intervals composing \( I(e) \) which label \( e \). The compactness of an IRS \( \mathcal{R} \) on \( S \), denoted by \( \text{Comp}(\mathcal{R}) \), is the maximum, over all arcs \( e \in E \), of the compactness \( c(I(e)) \). The IRS is \( k \)-IRS if, for every arc \((u, v)\), the collection of labels \( I(u, v) \) assigned to it is composed of at most \( k \) intervals \([a, b]\) of consecutive integers, where \( n \) and 1 being considered consecutive (cyclically.)

For general graphs, the problem of deciding whether \( \text{IRS}(S) = 1 \) is NP-complete.

The compactness of many classes of graphs has been studied. The trees, outerplanar graphs, hypercubes and meshes have compactness = 1. The Peterson graph has compactness = 2. Clearly, the compactness cannot exceed \( n/2 \), since any set \( I(e) \subset \{1, 2, \ldots, n\} \) containing more than \( n/2 \) integers must contain at least two consecutive integers, which can merge into an interval. Gavoille and Peleg [GP99] have shown that \( n/4 \) is asymptotically a tight bound for the compactness of \( n \)-node graphs.

The stretch factor represents the ratio between the length of a path and the distance between the endpoints. The maximum and the average stretch factors of \( \mathcal{R} \) are defined as follows:

\[
\text{Stretch}_G(\mathcal{R}) = \max_{x \neq y} \frac{\text{dist}_G(R;x,y)}{\text{dist}_G(x,y)}
\]

\[
\text{AvStr}_G(\mathcal{R}) = \frac{1}{n(n-1)} \sum_{x \neq y} \frac{\text{dist}_G(R;x,y)}{\text{dist}_G(x,y)}
\]

A routing scheme of stretch factor 1 is called a shortest path routing scheme.
The efficiency of a routing scheme is measured by the *stretch factor*, but the memory required in each node depends on the *compactness*. Intuitively, smaller compactness and degree imply smaller routing tables.

### 3.3 Pivot Interval Routing Scheme

Eilam [EGP98] defined the PIR as a particular case of the IRS, with the stretch factor at most five and the average stretch at most three. The scheme is simple and the paths are loop-free. The idea is to choose a collection of special nodes called *pivots* and to partition the network such that each set of the partition contains only one pivot. The pivot will be the main router for its set and the communication path between two nodes depends whether the nodes are in the same set or not. Each pivot relabels the node in its partition in ascending order, using the *tree-labeling* method (*DFS* for selecting the nodes.) The first pivot starts with value 1. If $n_1$ is the last value used by the first pivot for relabeling its nodes, the next pivot starts with $n_1 + 1$ for itself, and proceeds similarly for its clients, so on.

Now, each node, including the pivots, will have to relabel the outgoing arcs with the labels for destination nodes. It starts with the nodes which are successors in the partition, the nodes with are closer in terms of distance, and the pivots. The nodes which are further, in other partitions, are grouped, and for them we consider as the destination their pivots. In this way, each link is labeled with at most $2\sqrt{n(1+\ln(n))}$ intervals.

As a result of relabeling, when the nodes are in the same set, the messages will always follow the shortest path. If the sender and the destination are in different sets, the message sent by the sender will reach the pivot of the set where the destination is on the shortest path. Then, the message is sent on the shortest path from that pivot to the destination.

For every node $v$, we define an order relation among the nodes $\prec_v : x \prec_v y$ iff either $\text{dist}(x,v) < \text{dist}(y,v)$ or $\text{dist}(x,v) = \text{dist}(y,v)$ and $\text{ID}(x) < \text{ID}(y)$.

The routing scheme construction is based on the notions of balls and covers.

**Definition 3.3.1 (T-Ball)** We can order all the other nodes with respect to this relation and choose the set $t$-ball $B_v(t)$ as the first $t$ nodes according to the node ascending ordering.
Therefore, the t-ball defines the closer nodes, and does not always contain all the neighbors of the current node.

Consider now a collection $H$ of subsets of size $t$ of elements from $V$.

**Definition 3.3.2 (Cover)** A set $P \subseteq V$ is called the cover for $H$ if for every set $A \in H$, $A \cap P \neq \emptyset$.

Awerbuch [ABNLP90] presented two techniques to calculate the cover. The first technique uses a *greedy* algorithm: starting initially with $P = \emptyset$, adds iteratively an element of $V$ occurring at the most uncovered sets to $P$. When $P$ becomes a cover, the algorithm stops. The set $P$ is called a *greedy cover* for $H$.

Another technique uses *randomization*. Each element of $B$ is selected in the set $P$ with a probability $c \ln(H)$, for some constant $c > 1$. $P$ is called a *randomized cover* for $H$.

### 3.3.1 Overview of PIR

The PIR strategy, as described in [EGP98], has a preprocessing and a delivery part. The purpose of the preprocessing part is to gather information and to build the routing tables in each node. The delivery protocol simply takes the message $M$ with destination $v \neq u$ and sends it on the unique arc $e \in E_u$ such that $L(v) \in I(v)$.

**a. Preliminary construction:**

1. Let $P$ be a cover for the collection of t-balls of the nodes, $\{B_v(t) \mid v \in V\}$.
2. For $\forall v \in V$, $p(v)$ the nearest node to $v$ in $P$ with respect to the $\prec_v$ relation.
3. For every pivot $p \in P$, let $S_p = \{v \mid p(v) = p\}$ be the set of its "clients".
4. For every pivot $p \in P$, construct a minimum spanning tree $T_p$ rooted at $p$ and spanning the entire graph $G$. and let $T'_p$ be the subtree of $T_p$ induced by $S_p$.

**b. Labeling the nodes:**

Assume that $P = \{p_1, p_2, \ldots, p_l\}$. We start by labeling the nodes in $S_{p_1}$, then $S_{p_2}$, so on. We assign the nodes of $T'_{p_1}$ a pre-order numbering in ascending order, starting
from 1. Assuming $n_1$ is the largest label assigned to a node in $S_{p_1}$, we continue with the value $n_1 + 1$ to relabel the nodes in $S_{p_2}$ traversing the subtree $T'_{p_2}$, so on.

c. Labeling the arcs:

For a node $u \in V$, we label every arc $e \in E_u$ by a set of destinations $I(e) \subseteq V$ in three steps, starting with $I(e) = \emptyset$:

1. if $u$ is not a leaf in $T'_{p(u)}$, all its successors $s \in T'_{p(u)}$ label the appropriate arcs of $E_u$ on the shortest path from $s$ to $u$
2. for all the nodes $v \in B_u(t)$, label the appropriate arcs of $E_u$ on the shortest path from $u$ to $v$
   for all the pivots $p \in P$, the appropriate arc $e \in E_u$ on the shortest path from $u$ to $p$ is labeled by all the nodes in $S_p$ which are not part of the $B_u(t)$.

For example, consider the following arbitrary network:

![Figure 3.1: An arbitrary network](image)

The number of nodes $n = 10 \Rightarrow t = \lfloor \sqrt[10]{10(1 - \ln 10)} \rfloor \Rightarrow t = 5$

So each t-ball will have five elements, in case we have at least six nodes:

$B_t(v) = B_t(5), \forall v \in \{1, 4, 5, 7, 8, 9, 10, 11, 12, 13\} (6 = 5 + 1$ which is the current node, not included in its t-ball.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A t-ball is calculated using the order relation, based on the distance to the current node, defined at the beginning of Section 3.3.

From the Figure 3.1, the t-ball $B_t(5)$ is:

![Figure 3.2: The t-ball $B_t(5)$](image)

and the nodes from $B_t(5)$ are shadowed. We have:

<table>
<thead>
<tr>
<th>node</th>
<th>12</th>
<th>7</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

We observe that nodes 5 and 8 have the same distance to node 1, but the ID of node 5 is lower than the ID of node 8, so 5 has priority:

$B_t(5) = \{12, 7, 5, 8, 10\}$

For other nodes:

<table>
<thead>
<tr>
<th>node</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>13</th>
<th>1</th>
<th>9</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

and again 12 and 13 have the same distance, but 12 < 13. Also, here we have another situation: the nodes 1, 9, 11 have the same distance 9, but because we can select only one extra node, 1 is included and 9, 11 are rejected.
Based on the t-balls, we calculate the cover, applying a Greedy method. The frequency of the nodes in the t-balls (how often each node appears in a t-ball) are:

<table>
<thead>
<tr>
<th>node</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
</tr>
</tbody>
</table>

The nodes for the cover are selected based on high frequency, and in case of tie, we select the lesser ID. So, the first node selected is 10, then the t-balls containing 10 are eliminated and the frequency for all the nodes in the remaining t-balls (B₅, B₇, B₁₀) are computed again. The new values are:

<table>
<thead>
<tr>
<th>node</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
</tr>
</tbody>
</table>

so the node 9 is the second pivot, and the selection stops, because we do not have any remaining t-balls.

The cover is: cover = \{9, 10\} and, based on the distance, we have the following set of clients for each pivot:
- nodes 1, 4, 8 have their pivot = 10 because 10 is the only pivot included in their t-balls $B$
- nodes 5, 7 have their pivot = 9 because 9 is the only pivot included in their t-balls $B$
- node 9 has its pivot = 9 even if $10 \in B_9(5)$. because 9 is a pivot node.
- node 10 has its pivot = 10 even if $10 \in B_{10}(5)$. because 10 is a pivot node.
- node 11 has its pivot = 9 even if both $9, 10 \in B_{11}(5)$. because $\text{dist}(11, 9) < \text{dist}(11, 10)$.
- node 12 has its pivot = 10 even if both $9, 10 \in B_{12}(5)$. because $\text{dist}(12, 10) < \text{dist}(12, 9)$.
- node 13 has its pivot = 9 even if both $9, 10 \in B_{13}(5)$. $\text{dist}(13, 9) = \text{dist}(13, 10)$. but $9 < 10$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 4

THE DISTRIBUTED PIVOT INTERVAL ROUTING SCHEME

We present a fault-tolerant distributed implementation of the PIR scheme in a general asynchronous network. The algorithm starts with no knowledge of the network architecture and progressively builds an IRS scheme. It supports faults causing nodes and link failures and additions of nodes and/or links, and it is guaranteed that it will reach a correct state in finite time (it is self-stabilizing.) The tasks are fairly distributed among the nodes and each node builds its own routing table based on the information gathered online up to the current moment. There are nodes in the network (called pivots) which play the role of the routers, but their routing table capacity is the same as that of non-pivot nodes. The stretch factor (the ratio between the length of a path and the distance between the endpoints) is at most five and is three on the average. The stabilization time of the algorithm is $O(d \sqrt{n(1 + \log n)})$ time units, where $n$ is the number of nodes and $d$ is the diameter of the network.

In this chapter we present the self-stabilizing distributed algorithm. We start by defining new concepts as partial $t$-ball and nearer pivot, and a new technique called fair coordination, followed by the topology protocol requirements. The data structures and the global variables used in the code are presented in Section 4.1. Starting with the Section 4.2, we present the general structure of the algorithm, and then we give details, layer by layer.

We cannot give a bound on the moment of time when a message can be received, since the system is asynchronous. Also, we cannot wait forever to receive all the messages sent by other nodes in order to construct a correct $t$-ball and later, a correct cover. Eventually, after an unbounded period of time, all the messages will be received by the nodes and $t$-balls
will be correctly calculated and used in the algorithm. Therefore, we relax the definition of t-ball and nearest pivot to fit to an asynchronous algorithm:

**Definition 4.0.3 (Partial T-Ball)** A partial t-ball for a node $v$, $B_v$, is a set of $t$ nodes, with the length of the path toward $v$ within the $t$ lowest values received by the node until a certain condition becomes true.

In the description of the algorithm, we use the word $t$-ball instead of partial $t$-ball, by a slight abuse in notation.

Also, each non-pivot node (a node which is not part of the cover) has to choose its nearest pivot, with respect to the distance relation defined in Section 3.1. Because the network is an asynchronous system, we cannot bound the point in time when that pivot is correctly chosen. We define and we use the term nearer pivot, instead of closest pivot, in our asynchronous algorithm.

**Definition 4.0.4 (Nearer Pivot)** A nearer pivot for a node $v$ is a pivot $p \in P$, which is in the partial t-ball of the node $v$, $B_v$, and also with the lowest length of the path toward $v$ received until a certain condition becomes true.

For the distributed algorithm, we define a new technique called fair-coordination among the nodes of an ordered set $P$:

**Definition 4.0.5 (Fair-Coordination)** If $P = \{p_1, p_2, \ldots\}$ is an ordered set of nodes. $P \subseteq V$, we say that these nodes are fair-coordinated if each node $p_i$ starts executing its algorithm when it is its turn (this means that node $p_1$ starts first, node $p_2$ waits for $p_1$ to finish and then starts executing, and so on.)

Later, we will see that $P$ is in fact a greedy cover for all the t-balls of the nodes in the network.

We consider that each network link delivers every message sent in FIFO order to the receiver. In [APSV91] it is shown how to implement such a FIFO link in a self-stabilizing manner. Also, we consider an underlying topological maintenance protocol which is a local topology protocol. It assures properties regarding
-the message's sending/delivering: FIFO order in message receiving, no message loss (between neighboring processes), correct delivery of messages based on their types (for receive guards.)

-and the status of the links and neighboring nodes: the topology layer detects any modification in the adjacent links and continuously updates the values for the set of the up neighbors, the degree of the node and the set of all up nodes in the network. In every node it stores these current values in some variables.

In the same node we use shared memory between the processes (running algorithms), and message passing between any two neighboring nodes. Each algorithm communicates by global variables with the other processes running in the same node. The global variables are described in the next subsection.

4.1 Data Structures

Each node \( u \) maintains several variables of different types. One variable is computed by the underlying layer of topological maintenance protocol:

\[ N_v = \text{the set of the neighbors' IDs of the node } v \]

\( leader = \text{the elected leader for all the nodes in the graph} \)

\( to.leader = \text{the neighbor toward the leader} \)

The others are calculated and used by the layers of the algorithm. The most important ones are:

i) integer variables:

\( t \) is the dimension of a t-ball and is equal to \( \sqrt{n \cdot \ln n} \)

\( pivot \) contains the nearest pivot

\( to.pivot \) the neighbor which is the first node on the shortest path to that particular pivot

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ii) Boolean variables:

exist_leader is true when the leader-node is selected

updated is true when the t-ball is updated

ball_sent is true when the t-ball is sent to the leader

cover_ready is true when the node has received the cover

relabel is true when a node has selected its pivot/clients and is ready for the relabeling step

reset2, reset3, reset4 are true when we restart the algorithm for the layer 2, 3.

iii) other data structures:

L, I the node, respectively the arc relabeling functions

B a set of the integer values corresponding to the nodes IDs situated in the t-ball B_v of the node v

Recvd.IDs the set of IDs of other nodes known by the current node

cover a set of integer values representing the IDs of the pivots which forms the cover for all the t-balls

S a set of the integer values corresponding to the successors IDs situated in the subtree S_p rooted in the closest pivot p of the node

SPT = the shortest path tree rooted at itself, if the node is elected as a pivot

H a linked list with information regarding the nodes from B, cover and S.

Each element of the list has three fields of type int and the last one of type Boolean:

dest = destination ID

neighbor = the neighbor which is the first node on the path to dest

distance = the distance toward dest, or 0
**direct** = is true if **dest** is a direct neighbor of the current node and the
direct link is the shortest

The list **H** is maintained in ascending order by the value of the destination and has several
functions which help us to retrieve information from it:

**Give.IDs(H)** = returns all the nodes IDs stored in the field **id** in **H**. or null
if **H** is ∅

**G(H,id)** = returns the element of **H** with the given **id**. or null if **H** does not
have such an element

We consider the following notations:
- v ∈ **H** means v ∈ **Give.IDs(H)** (e.g. H ⊈ (B ∪ cover ∪ S) means **Give.IDs(H)** ⊈
(B ∪ cover ∪ S))
- **H[id]** means G(H.u) (e.g. H[u].neighbor means (G(H.u)).neighbor.)

4.2 The Algorithm

We present the layers of the algorithm, with an informal explanation for each layer and
the subsequent module running on that particular layer.

The algorithm has four layers:

**First layer:** Alg. Calculate.Ball

**Second layer:**
Alg. Receiving.Cover
or
Alg. Calculate.Cover
- all nodes except the leader -

**Third layer:**
Alg. To.Clients
or
Alg. To.Pivots
- the pivots from cover -

**Fourth layer:**
Alg. Pivot.Label
or
Alg. Client.Label
- the pivots from cover -

- the non-pivots -
The general algorithm is:

\texttt{Algorithm 4.2.1 SPIR \ Stabilizing Pivot Interval Routing Scheme}

\begin{itemize}
  \item[A.01] \textit{Error Correction}
  \item[A.02] \textit{Calculate Ball}
  \item[A.03] \textit{Receiving Cover}
  \item[A.04] \textit{Calculate Cover}
  \item[A.05] \textit{To Clients}
  \item[A.06] \textit{To Pivots}
  \item[A.07] \textit{Pivot Label}
  \item[A.08] \textit{Client Label}
\end{itemize}

\textit{Error Correction} has the role to broadcast periodically a message \textit{DIST} to inform about the node. Eventually discrepancies in the global variables are detected and appropriate actions are taken. Because it does not participate in the process of constructing the IRS, it is not included in any of the layers.

In the first phase, the algorithm \textit{Calculate Ball} gathers information about the neighborhood. The \textit{DIST} messages from the other nodes are processed and the first \( t \) lowest distances, breaking ties by increasing node ID, are stored in the data structure \( H \) together with the node IDs (stored also in the set \( B \).) So far, \( B_c \) is computed in the set \( B \) and the corresponding links are labeled dynamically with the IDs from the set \( B \). The leader is elected and we know the neighbor toward it (each node is oriented toward the leader by the variable \textit{to.pivot}.) When we have received enough information, \( B \) is sent to the leader.

The modules in the second phase concentrate in broadcasting all the \( t \)-balls toward the leader, calculating and broadcasting the cover of all the \( t \)-balls to the nodes in the network. The leader will execute the algorithm \textit{Calculate Cover}, which, using a greedy method, calculates the set of the pivots. The cover is broadcast in the network such that now a node can see if it is elected as a pivot or has to be a client for one particular pivot. From a node, the message containing the \( t \)-ball follows the shortest path toward the leader.
passing by intermediate nodes (Receiving_Cover.) Later, from the leader, the cover is sent to all the nodes. An intermediate node has to analyze each message received and to forward it on the appropriate link.

The third phase partitions the network into sets of clients, one set for each pivot. Once the set of pivots (the cover) is known by each node, each pivot has to create its own 'clienthood'. In order to attract clients, the pivots send messages in the network (algorithm To_Clients) and leave the clients to choose the nearest pivot (algorithm To_Pivots.) And each node relabels the appropriate links toward the pivots, and its successors in the clienthood.

The last phase consists in deciding (algorithm Pivot_Label) and in assigning new IDs (algorithm Client_Label) for the nodes such that, in each node, the destinations can be grouped into a set of intervals labeling the links to the neighbors. Using fair-coordination, described at the beginning of Section 4, each pivot relabels the nodes in its partition, and once this is done, it broadcasts the result such that the next pivot can start. Each node in the network receives, gradually, the result of relabeling. Once a result is received, a non-pivot node changes the IDs appropriately. For a pivot, the relabeling follows the tree-labeling algorithm applied to the spanning tree containing all of the pivot's clients. The method consists of relabeling the nodes in a preorder traversal of the tree, using correct values (values from the correct range.)

4.2.1 Error_Correction

The algorithm Error_Correction in each node checks continuously the values of the global variables to detect erroneous values and sends periodically messages to the other nodes in order to help them calculate the distance to the current node. If an error is detected, the entire construction of the SPIR scheme must start from scratch. In this case, all the global variables are reset to null or false, in order to start a fresh phase.
Algorithm 4.2.2 Error Correction

Messages: 

- **DIST**: sender: the ID of the sender
dist: the length of the path the message went through
- **LOST**: id1: the sender IDid2: the ID of the other node adjacent to the crashed link

Local variables: 

- id, nb, nbr: int

Predicate error ≡ (B \ H \neq \emptyset) \lor (B = \emptyset \land (H \cup cover \cup S) \neq \emptyset) \lor (cover = \emptyset \land S \neq \emptyset) \lor (H \neq \emptyset \land (H \cup cover \cup S) \neq \emptyset) \lor (to.pivot \notin \{N_v \cup \{ID_v\}\})

Macro: **RESTART** = reset all the layers and set the global variables to their default values

Actions:

1.01 timeout → /* node v broadcasts the message DIST */
1.02 SEND DIST(ID_v, 0) TO all nb \in N_v
1.03 error → **RESTART**
1.04 \exists id \in H : H[id].direct = true \land id \in N_v \land \text{length(link to id) \neq H[id].distance} →
1.05 **RESTART**
1.06 \exists id \in H : H[id].direct = false \land id \in N_v \land H[id].neighbor = id →
1.07 H[id].distance := \text{length(link to id)}
1.08 H[id].direct := true

Macro **RESTART**

- R.01 exist_leader.ball_sent.cover.ready, relabel := false
- R.02 B, H, cover, S := \emptyset
- R.03 pivot := ID_v
- R.04 Recvd.IDs := \{ID_v\}
- R.05 to.pivot := some neighbor from N_v
- R.06 reset2, reset3, reset4 := true

4.2.2 Calculate Ball

The set B should contain t nodes with the lowest t distances to v. H should contain all the nodes in the graph, divided into intervals, and, for each interval, the neighbor of v toward them.

When B contains the t lowest distances among all received, it is sent to the leader. Whatever is stored in B, is stored also in H, with additional information regarding the distance to those nodes and the neighbors of v toward them, so we have further tests on H instead of B. At the same time, each node sends its t-ball to each neighbor in order to detect eventual discrepancies. From [EGP98], we know that:

If u \in B_v then for every node x on the shortest path from v to u, u \in B_x.
In our algorithm, this is the way a node \( v \) selects its nodes in \( B \) by comparing different distances received from all the neighbors which also includes those nodes in their t-balls. Thus, checking the other t-balls help us to eliminate wrong nodes and cycles in delivering messages.

On receiving a message \( DIST \) from a neighbor \( nbr \):
- if the message contains its own ID (\( DIST.sender = ID_v \)), discard it
- otherwise, process the message and eventually broadcast it to the other neighbors (if any.)

Message processing means:
- add the length of the link to \( nbr \) at the field distance in the message
- if the updated distance is within the top of the \( t \) lowest distances, breaking ties by increasing node ID. the ID is stored in \( B \) and \( H \). and the message will be broadcast to all the other neighbors.
- otherwise, discard the message.

When \( v \) has received at least one message from all the other nodes in \( V \). and later, whenever a message received changes the data structure \( B \) (and \( H \)). the partial t-ball \( B \) is sent to the leader using the neighbor \( to.leader \).

---

**Algorithm 4.2.3 Calculate_Ball**

**Messages:**
- \( BALL \): sender: the ID of the sender
  - \( B \): the set \( B \)
  - \( dest \): the ID of the destination
- \( CHECK \): sender: the sender ID. a neighbor of the current node
  - \( BN \): the t-ball of the sender
  - \( HN \): the data structure \( H \) of the sender
- \( DIST \): sender: the ID of the sender
  - \( dist \): the length of the path the message went through
- \( LOST \): \( id_1 \): the sender ID
  - \( id_2 \): the ID of the other node adjacent to the crashed link

**Local variables:**
- \( id. u. nb. nbr \): int /* elements in \( N_e */
- \( updated.to.send \): Boolean

**Macros:**
- \( REMOVE \) = eliminate a wrong node and restart the layers 2.3 and 4
  - \( input \): id: int /* the wrong node to be removed */
  - \( N \): set of int /* the set of neighbors to be warned about */

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.2.4 Calculate_Ball

- **RESTART** = restart the layers 2.3 and 4
- **SEND_NBRS** = send DIST messages to a set of neighbors and update some local variables
- **UPDATE** = update the data structure $B$ and $H$  

**Actions:**

1. **ID** = leader $\rightarrow$ exist.leader := true
2. $(B \cup \{ID_v\}) \setminus$ Rcvd.IDs $\neq \emptyset$ $\rightarrow$ RESTART
3. $\exists id \in H : (id \notin N_v \land H[id].direct = true) \lor H[id].neighbor \notin N_v$ $\rightarrow$ 
   - /* id is a wrong node and remove it from $B$ and $H */
   - REMOVE(id \in N_v)
4. Rcvd.IDs := \{ID_v\}
5. **UPDATE** = update the data structure $B$ and $H$
6. **RESTART**

Upon RECEIPT of $DIST(s, dist_s)$ FROM neighbor nbr $\rightarrow$

1. if $(s \neq ID_v)$
2. dist_s := dist_s + length(link to nbr)
3. /* update the information in $B, H */
4. **UPDATE** $(s, dist_s, nbr)$

Upon RECEIPT of $LOST(id_v, id_j)$ FROM nbr $\rightarrow$

1. if $(id_v \in H \land H[id_j].neighbor = id_v)$
2. REMOVE(id_j \in N_v \setminus \{nbr\})

$(\#Rcvd.IDs = n \land$ updated $\land$ exist.leader) $\rightarrow$

1. if $(ID_v \neq$ leader)  
2. **SEND** BALL$(ID_v, B, leader)$ TO to.leader
3. **endif**
4. ball.sent := true
5. updated := false

$(\#Rcvd.IDs = n \land$ to.send) $\rightarrow$

1. **SEND** CHECK$(ID_v, B, H)$ TO all $nb \in N_v$
2. to.send := false

Upon RECEIPT of $CHECK(s, B_s, H_s)$ FROM nbr $\rightarrow$

1. if $(s = nbr)$
2. **then**
3. for all $(u \in B \land u \neq s \land H[u].neighbor = s)$ do
4. if $(u \notin B_s) \lor (u \in B_s \land H[u].distance \neq H_s[u].distance + length(link to nbr))$
5. **then**
6. REMOVE(u \in N_v)
7. **endif**
8. **endfor**
9. **endif**
Algorithm 4.2.5 Calculate_Ball

Macro REMOVE(id, N)
R.01 RESTART;
R.02 Remove_ID(id, H, B)
R.03 if (to.pivot = id)
R.04 then
R.05 to.pivot := some neighbor from N
R.06 endif
R.07 SEND LOST(IDs, id) TO all nb ∈ N
R.08 Revd.IDs := Revd.IDs \ {id}

Macro RESTART;
T.01 ball.sent. cover.ready. relabel := false
T.02 reset2, reset3, reset4 := true
T.03 cover, S := Ø
T.04 updated, to.send := true

Macro SEND_NBRS
/* the message DIST is forward to the other neighbors */
S.01 SEND DIST(s.dist, s) TO all nb ∈ N \ {nbr}
S.02 updated, to.send := true

Macro UPDATE(id, dist, nbr)
Local.variables: ID.max.dst : int
U.01 Revd.IDs := Revd.IDs ∪ {s}
U.02 if (id = leader)
U.03 then
U.04 exist.leader = true
U.05 endif
U.06 if (id ∈ B)
U.07 then
U.08 if (id ∈ H ∧ (H[id].distance > dist ∨ (H[id].distance < dist ∧ H[id].neighbor = nbr)))
U.09 then
U.10 H[id].distance := dist
U.11 H[id].neighbor := nbr
U.12 H[id].direct := false
U.13 SEND_NBRS
U.14 endif
U.15 else
U.16 if (|B| < t)
U.17 then
U.18 NewCell(H, B, id, dist, nbr, false)
U.19 SEND_NBRS
U.20 else
U.21 ID.max.dst := Maximum_Distance(B, H)
U.22 if (H[ID.max.dst].distance > dist) ∨ H[ID.max.dst].distance = dist ∧ ID.max.dst = id)
U.23 then /* id is inserted and ID.max.dst is removed */
U.24 RESET(ID.max.distance, N)
U.25 NewCell(H, B, id, dist, nbr, false)
U.26 SEND_NBRS
U.27 endif
U.28 endif
U.29 endif
4.2.3 Partial Algorithms - Second phase

1. Routing the t-balls or the cover (each node except the leader)

A node $v \neq \text{leader}$ is waiting for the calculated cover from the leader. Eventually messages containing other t-balls are forwarded to the leader. Once the cover reaches the node $v$ through a neighbor, $v$ broadcasts it to the other neighbors.

Algorithm 4.2.6 Receiving_Cover

<table>
<thead>
<tr>
<th>Messages:</th>
<th>( \text{BALL:} )</th>
<th>( \text{sender} : ) the ID of the sender</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( B : )</td>
<td>( \text{set B} )</td>
</tr>
<tr>
<td></td>
<td>( \text{dest} : )</td>
<td>( \text{ID of the destination} )</td>
</tr>
<tr>
<td>( \text{COV:} )</td>
<td>( \text{sender} : )</td>
<td>( \text{ID of the sender} )</td>
</tr>
<tr>
<td></td>
<td>( \text{cover} : )</td>
<td>( \text{cover (set of pivots)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local ( \text{variables:} )</th>
<th>( \text{id, dest, nbr} : ) int /* elements in ( N_v */</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B, \text{cov} : )</td>
<td>( \text{set of t int} )</td>
</tr>
</tbody>
</table>

| Predicate: \( \text{wait.cover} \equiv \text{ball.sent} \land \text{ID} \neq \text{leader} \) |

| Macro: \( \text{RESTART}_2 \) | \( \equiv \) restart the layers 3 and 4 |

<table>
<thead>
<tr>
<th>Actions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01 ( \text{wait.cover} \land \text{reset2} = \text{true} )</td>
</tr>
<tr>
<td>1.02 Upon RECEIPT of ( \text{BALL}(s, B, \text{dest}) ) FROM ( \text{nbr} )</td>
</tr>
<tr>
<td>1.03 if ( (\text{ID} \neq \text{leader} \land s \neq \text{ID} \land \text{nbr} \neq \text{to.leader} \land \text{dest} = \text{leader}) )</td>
</tr>
<tr>
<td>1.04 then ( \text{why a message to be forwarded to the leader} )</td>
</tr>
<tr>
<td>1.05 SEND ( \text{BALL}(s, B, \text{dest}) ) TO ( \text{to.leader} )</td>
</tr>
<tr>
<td>1.06 endif</td>
</tr>
<tr>
<td>1.07 Upon RECEIPT of ( \text{COV}(s, \text{cov}) ) FROM ( \text{nbr} )</td>
</tr>
<tr>
<td>1.08 if ( (\text{wait.cover} \land s = \text{leader} \land \text{nbr} = \text{to.leader} \land \text{cover} \neq \text{cov}) )</td>
</tr>
<tr>
<td>1.09 then ( \text{why a new cover from the leader} )</td>
</tr>
<tr>
<td>1.10 ( \text{cover} := \text{cov} )</td>
</tr>
<tr>
<td>( \text{why broadcast the cover to the other neighbors} )</td>
</tr>
<tr>
<td>1.11 SEND ( \text{COV}(s, \text{cov}) ) TO ( \text{nb} \in N_v \setminus {\text{nbr.to.leader}} )</td>
</tr>
<tr>
<td>1.12 ( \text{cover.ready} := \text{true} )</td>
</tr>
<tr>
<td>1.13 ( S := \emptyset )</td>
</tr>
<tr>
<td>1.14 endif</td>
</tr>
</tbody>
</table>

Macros \( \text{RESTART}_2 \)

| R.01 \( \text{cover.ready, relabel} := \text{false} \) |
| R.02 \( \text{reset3, reset4} := \text{true} \) |
| R.03 \( \text{cover}, S := \emptyset \) |
| R.04 \( \text{reset2} := \text{false} \) |

2. Calculate the cover (only the leader)

The leader receives all the t-balls \( B_v(t), v \in V \), and using a greedy algorithm decides the cover (the set of the pivots.) The result is broadcast in the network.
Algorithm 4.2.7 Calculate_Cover

Messages: BALL: sender : the ID of the sender  
B: the set B  
dest : the ID of the destination  
COV: sender : the ID of the sender  
cover : the cover (set of pivots)

Local.variables: update:boolean  
Rcvd : set of int  
TBalls : Linked.List of (int. set of t int)

Predicate: determine.cover \equiv ball.sent \land ID_\pi = leader

Macro: UPDATE.TBALLS = update the current set of tballs by adding new or replacing the old ones

input: id : int  
B.d : set of t int  
TBalls : Linked.List of (int. set of t int)

Actions:

401 \quad determine.cover \land reset2 = true \quad \rightarrow \quad /* eliminate all the t-balls received */
402 \quad update := false
403 \quad Rcvd, TBalls := \emptyset
404 \quad RESTART;

405 \quad Upon RECEIPT of BALL(s, B, dest) FROM nbr \quad \rightarrow
406 \quad if (ID_\pi = leader \land dest = ID_\pi)
407 \quad then
408 \quad UPDATE.TBALLS(s, B, TBalls)
409 \quad endif

410 \quad determine.cover \land |Rcvd| = n \land update \quad \rightarrow
411 \quad /* we can calculate the cover for all the t-balls including the local one B */
412 \quad UPDATE.TBALLS(ID_\pi, B, TBalls)
413 \quad cover := Greedy_Cover(TBalls)
414 \quad /* the node sends the cover to all neighbors */
415 \quad SEND COV(ID_\pi, cover) TO all nb \in N_\pi
416 \quad update := false
417 \quad cover.ready := true
418 \quad S := \emptyset

Macro UPDATE.TBALLS(id, B,d, TBalls)
\quad 01 \quad Rcvd := Rcvd \cup \{id\}
\quad 02 \quad if (id \in Give.IDs(TBalls))
\quad 03 \quad then
\quad 04 \quad Remove_Ball(id, TBalls)
\quad 05 \quad endif
\quad 06 \quad New_Ball(id, B,d, TBalls)
\quad 07 \quad update := true

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
4.2.4 To-Clients and To-Pivots

From this point, the nodes are divided into pivots and non-pivots. A node runs either the algorithm To.Pivots or To.Clients. Both modules take in consideration the fact that any node, pivot or client, must have links oriented toward each pivot.

A non-pivot node has to choose a pivot, the nearest one, and to become the client of that pivot. It has to know the distance to each pivot, therefore it waits for the pivots' action:

\[ B_v \cap \text{cover} \neq \emptyset \implies \exists \text{ a pivot in } B_v \]

At the same time, each pivot \( p \in \text{cover} \), receiving the set of pivots, has to identify the set of its clients. So \( p \) will broadcast a message \( P\_DIST \) containing its ID and the distance, initially 0, to every other node. Such messages will be received by every node, pivot or not, after the set of the pivots, the \( \text{cover} \), is already known. They will help each node to store the first node on the shortest path to each pivot, and non-pivot nodes to decide their nearest pivots.

So, a node (pivot or not), receiving \( P\_DIST \) from a pivot through a neighbor \( \text{nbr} \), does the following:

- if the message ID is its own ID, discard the message
- otherwise, add the length of the link to \( \text{nbr} \) at the field \( \text{distance} \) in the message and continue.
- if, for that particular node ID, it is the first message received or the updated distance is less than the stored one, memorize for the link the ID of the message in data structure \( H \)
- otherwise, discard the message.

1. **Calculate and organize the set of the clients (each pivot \( p \in \text{cover} \)**

A pivot receives another type of messages, the answers of the clients (\( \text{ACCEPT\_Succ} \) or \( \text{REJECT\_Succ} \)). In case of \( \text{ACCEPT\_Succ} \), the sender is added to the data structure \( H \), with distance 0. Also, the \( \text{successors} \) field represents a path in the \( \text{SPT} \) rooted at pivot:
- last ID is the "son" of the pivot
- the next last is the "son of the son", and so on.

and it is added to the tree.

---

**Algorithm 4.2.8 To.Clients**

**Messages:**
- `ACCEPT.SUCC`: `sender`: the ID of the sender
  `succs`: the string of successors' IDs
  `dest`: the ID of the destination
- `P.DIST`: `sender`: the ID of the sender
  `dist`: the length of the path the message went through
- `REJECT.SUCC`: `sender`: the ID of the sender
  `dest`: the ID of the destination

**Local.variables:**
- `nb nbr`: int /* elements in \( X_v \) */
- `Rcvd`: set of int

**Predicate:**
- `select.clients \( \subseteq \) cover.ready \& 1D_\{i\} \in cover`

**Macros:**
- `ALLANSWERS`: check whether the pivot has received messages from everybody, in order to end this layer and to go to the next layer, for relabeling
  - `RESTART_\{i\}` = restart the layer \( i \)
  - `UPDATE.H` = update \( H \) information and returns `true` if a change was made, otherwise `false`

**Actions:**

1. \( \{ \) `select.clients \& reset3 = true` \( \) \( \rightarrow \)
2. `Rcvd := ID_v`
3. `RESTART_3`
4. \( \{ \) `select.clients \& timeout` \( \) \( \rightarrow \)
5. `SEND P.DIST(ID_v,0) TO all nb \in X_v`
6. `pivot := ID_v`
7. `to.pivot := ID_v`

8. Upon RECEIPT of `P.DIST(s.dist,) FROM nbr` \( \rightarrow \)
9. if `(select.clients \& s \neq 1D_v \& s \in cover)`
10. then /* update the data structure \( H \) */
11. `dist, := dist, + length(link to nbr)`
12. `UPDATE.H(s.dist,, nbr)`
13. `Rcvd := Rcvd \cup \{ s \}`
14. `endif`

15. Upon RECEIPT of `ACCEPT.SUCC(s.succ,, dest) FROM nbr` \( \rightarrow \)
16. if `(select.clients \& s \neq 1D_v \& dest = 1D_v)`
17. then /* add the sender as a client to \( H \) with the default distance 0 */
18. `UPDATE.H(id,0,nbr)`
19. /* message ACCEPT.SUCC contains a path in the SPT with only its own clients */
20. `Construct.Tree(SPT.succ,)`
21. /* check whether all the nodes have replied */
22. `ALLANSWERS`
23. `endif`
Algorithm 4.2.9 To_Clients

5.22 Upon RECEIPT of REJECT_SUCC(s, dest) FROM nbr
5.23 if (select_clients ∧ s ≠ ID_v)
5.24 then
5.25 if (dest = ID_v)
5.26 then
5.27 /* check whether all the nodes have replied */
5.28 ALL_ANSWERS
5.29 else
5.30 SEND REJECT_SUCC(s, dest) TO H[dest].neighbor
5.31 endif
5.32 endif

Macro ALL_ANSWERS
A 01 Rcvd := Rcvd ∪ {s}
A 02 if (|Rcvd| = n)
A 03 then /* go to the layer 4 */
A 04 relabel, reset4 := true
A 05 cover.ready := false
A 06 endif

Macro RESTART3
R 01 reset4 := true
R 02 S := ∅
R 03 relabel, reset3 := false

Macro UPDATE_H(id, dist, nbr)
U 01 if (id ∈ H)
U 02 then
U 03 NewCell(H, id, nbr, dist, false)
U 04 SEND P_DIST(id, dist) TO all nb ∈ N_v \ {nbr}
U 05 return true
U 06 else
U 07 if ((H[id].neighbor = nbr) ∨ (H[id].distance > dist))
U 08 then /* a new distance to that node */
U 09 H[id].distance := dist
U 10 H[id].neighbor := nbr
U 11 H[id].direct := false
U 12 SEND P_DIST(id, dist) TO all nb ∈ N_v \ {nbr}
U 13 return true
U 14 else
U 15 return false
U 16 endif
U 17 endif

2. Calculate the nearest pivot (each non-pivot v ∈ V - cover)

In addition, a non-pivot node has to do an extra action: if the updated distance is
the least one (this means that the message is from the nearest pivot), the node updates its
variables:

pivot ← that pivot
to.pivot ← the neighbor from which the message was received
and broadcasts the message to all the other neighbors (if any.)

When the node has received at least one message from each pivot, it has selected its nearest pivot and the pivot is in \( B \), the node sends back (using the \texttt{to.pivot} neighbor) a message \texttt{ACCEPT.SUCC} with its ID and the selected pivot as destination. The message follows a path containing nodes with the same pivot as its pivot.

A message from a non-pivot node must be an answer toward the same pivot as itself:
- the set of successors stored in the message is extracted and the neighbor, from which the message was received, is stored in data structure \( H \).
- the node adds its ID at the end of the message and forwards the message to the pivot using the \texttt{to.pivot} neighbor.

\begin{algorithm}
\caption{To.Pivots}
\begin{description}
\item[Messages:] \texttt{P.DIST:} \hspace{1em} \begin{description}
\item[sender:] the ID of the sender
\item[dist:] the length of the path the message went through
\end{description}
\item[REJECT.SUCC:] \hspace{1em} \begin{description}
\item[sender:] the ID of the sender
\item[dest:] the ID of the destination
\end{description}
\item[ACCEPT.SUCC:] \hspace{1em} \begin{description}
\item[sender:] the ID of the sender
\item[succs:] the string of successors' IDs
\item[dest:] the ID of the destination
\end{description}
\end{description}
\begin{description}
\item[Local.variables:] \hspace{1em} \begin{description}
\item[nb nbr:] int /* elements in \( V_r \) */
\item[min dist:] int /* the minimum distance to the current pivot */
\item[update:] boolean
\item[Rcvd:] set of int
\end{description}
\end{description}
\begin{description}
\item[Predicate:] \texttt{choose.pivot} \equiv \texttt{cover.ready} \land \texttt{ID} \not\in \texttt{cover}
\end{description}
\begin{description}
\item[Macro:] \texttt{BETTER.PIVOT} = update some global and local variables in order to choose the nearer pivot of the current node
\end{description}
\begin{description}
\item[Actions:] \hspace{1em} \begin{description}
\item[6.01] \texttt{choose.pivot} \land (\texttt{reset3 = true} \lor (\texttt{Rcvd \setminus cover} \neq \emptyset)) \rightarrow
\item[6.02] \texttt{Rcvd} := \emptyset
\item[6.03] \texttt{min.dist} := \infty
\item[6.04] \texttt{RESTART3}
\end{description}
\end{description}
\end{algorithm}
Algorithm 4.2.11 To.Pivots

Upon RECEIPT of $P\text{.DIST}(s,dist_s)$ FROM nbr  

\[ \text{if } (\text{choose.pivot} \land s \neq ID_v \land s \in \text{cover}) \]

\[ \text{then } /* \text{update the data structure } H */ \]

\[ \text{Recv} := \text{Recv} \cup \{s\} \]

\[ \text{dist}_s := \text{dist}_s + \text{length(link to nbr)} \]

\[ \text{if } (\text{pivot} \notin \text{cover}) \]

\[ \text{then} \]

\[ \text{pivot} := s \]

\[ \text{to.pivot} := \text{nbr} \]

\[ \text{min.distance} := \text{dist}_s \]

\[ \text{endif} \]

\[ \text{update} := \text{update} \lor \text{UPDATE.H}(s,dist_s,nbr) \]

\[ /* \text{check whether we can choose the nearest pivot */} \]

\[ \text{BETTER.PIVOT(min.distance, s.dist, nbr)} \]

\[ \text{endif} \]

\[ \text{choose.pivot} \land \text{Recv} = \text{cover} \land \text{update} \land \text{pivot} \in B \]

\[ /* \text{the closer pivot is selected, so } v \text{ sends acceptance to it and refusals to the other pivots */} \]

\[ \text{SEND ACCEPT.SUCC(ID_v.piv, pivot) TO to.pivot} \]

\[ \text{for all } pp \in \text{cover} \setminus \{\text{pivot}\} \text{ do} \]

\[ \text{SEND REJECT.SUCC(ID_v, pp) TO } H\{pp.neighbor} \]

\[ \text{endfor} \]

\[ /* \text{now activate the next layer */} \]

\[ \text{relabel, reset} := \text{true} \]

\[ \text{update, cover.ready} := \text{false} \]

Upon RECEIPT of REJECT.SUCC(s.dest) FROM nbr  

\[ \text{if } (\text{choose.pivot} \land s \neq ID_v \land \text{dest} \in \text{cover} \land \text{dest} \in H) \]

\[ \text{then } /* \text{a message for a pivot */} \]

\[ \text{SEND REJECT.SUCC(s.dest) TO } H\{\text{dest}.neighbor} \]

\[ \text{endif} \]

Upon RECEIPT of ACCEPT.SUCC(s.succ, dest) FROM nbr  

\[ \text{if } (\text{choose.pivot} \land \text{dest} \in \text{cover} \land \text{dest} = \text{pivot}) \]

\[ \text{then } /* \text{a message from a client of the same pivot */} \]

\[ S := S \cup \{s\} \]

\[ \text{UPDATE.H}(s, 0, nbr) \]

\[ /* \text{the node adds itself at the end of the message and forward it to the pivot */} \]

\[ \text{succ}_v := \text{succ}_v \pi \text{"+"} \text{ indicates the concatenation operator } ID_v \]

\[ \text{SEND ACCEPT.SUCC(s.succ, dest) TO to.pivot} \]

\[ \text{endif} \]

Macro BETTER.PIVOT(min.distance, id.dist, nbr)

\[ \text{if } ((\text{pivot} = \text{id} \land \text{to.pivot} = \text{nbr}) \lor (\text{min.distance} > \text{dist})) \]

\[ \text{then } /* \text{update the current pivot information */} \]

\[ \text{min.distance} := \text{dist} \]

\[ \text{pivot} := \text{id} \]

\[ \text{to.pivot} := \text{nbr} \]

\[ \text{endif} \]
4.2.5 PivotLabel and ClientLabel

Here we solve the most important part, relabeling the nodes.

1. Pivot-action

Each pivot relabels the nodes which are its clients when it is its turn. We use the new technique, fair coordination, defined in Definition 4.0.5:

- the first pivot in cover will start re-labeling; when this is done, the pivot broadcasts the message RELBL with the old and the new ID for each of its clients to all its neighbors in order to reach all the other nodes in the graph and subsequently, the other pivots

- once a message RELBL from the first pivot is received by the second pivot, it will proceed accordingly and so on. It accepts only one message from each other pivot (all the other are discarded.)

Each time a pivot starts relabeling, it warns the other nodes to be prepared to accept the (new) relabels, by sending in advance a CLEAN message. In the meantime, the messages containing new IDs are processed:

- the old node ID is replaced by the new one and the variables associated are updated

- the message RELBL is broadcast to all the neighbors, in the entire graph.

Algorithm 4.2.12 PivotLabel

Messages: RELBL: sender: the ID of the sender
  old.IDs: the old IDs of the nodes in S\textit{sender} which have received a new label
  new.IDs: the new labels of the nodes in S\textit{sender}

CLEAN: sender: the ID of the sender
  dest: the ID of the destination

Local.variables: V : set of int /* the set of current nodes in the graph */
  new.L: function from V to \([1..n]\) /* new node labeling function */
  new.A: function from E to \(2^\text{V}\) /* new arc labeling function */
  newlabels: boolean /* is true if at least one label is changed */
  n, ni, nbr, nb.newID: int
  Rcvd: set of int /* set of pivot IDs with the relabels already received */

Predicate: relabel клиентs \(\equiv\) relabel \(\land\) IDv \(\in\) cover

Macros: LABEL = the node v do the relabeling for all the nodes in its SPT tree S

REFRESH = send a message CLEAN to all other pivots regarding the new relabeling

RELABEL.DONE = check whether the new labels received are different from the previous ones and start a new phase

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.2.13 Pivot_Label

**RESTART** = restart the algorithm from the layer 2

**UPDATE_IDS** = build the interval routing scheme in the current node

**input:** current.id, pivot.id : int
oldIDs, newIDs : array of int

**Actions:**

1. **RESTART**
   - `relabel.clients ∧ (pivot ∉ cover ∨ H ≠ (B ∪ cover ∪ S \ {1D_c}))` → **RESTART**

2. **REFRESH**
   - `reset4 := true ∨ Rcvd ∩ cover ≠ ∅`
   - `reset4 := false`

3. **LABEL**
   - `relabel.clients ∧ (ID_v is the first pivot in cover)` → **REFRESH**

4. **Upon RECEIPT of CLEAN(s, dest) FROM nbr** →
   - `if (dest = 1D_v)`
   - `then /* a new start for some pivot */`
   - `Rcvd := Rcvd \ {s}`
   - `/* node v warns its clients also */`
   - `for all id ∈ S ∧ id ≠ 1D_v do`
   - `SEND CLEAN(1D_v, id) TO H[id].neighbor`
   - `endfor`
   - `else`
   - `SEND CLEAN(s, dest) TO H[dest].neighbor`
   - `endif`

5. **Upon RECEIPT of RELBL(s, oldIDs, newIDs) FROM nbr** →
   - `if (s ∈ cover ∧ s ≠ 1D_v ∧ s ∉ Rcvd)`
   - `then`
   - `V := V ∪ {oldIDs}`
   - `SEND RELBL(s, oldIDs, newIDs) TO all nb ∈ N_v \ {nbr}`
   - `/* update H corresponding to the new labels */`
   - `UPDATE_IDS(1D_v, s, oldIDs, newIDs)`
   - `/* store the last value assigned to a node */`
   - `n := MAX(newIDs)`
   - `if (n > ni)`
   - `then`
   - `ni := n`
   - `endif`
   - `/* check whether it is our turn */`
   - `if (∃pp ∈ cover before v and pp ∉ Rcvd)`
   - `then`
   - `LABEL`
   - `endif`
   - `endif`

6. `Rcvd = cover ∧ |V| = n` → **RELABEL_DONE**
Algorithm 4.2.14 Pivot_Label

Macro LABEL
/* the pivot has to do the relabeling for its clients starting with the next value for ni */
L.01 ni := ni + 1
L.02 (ni.old.IDs, new.IDs) := Tree_Labeling(IDv, SPT(IDv), ni)
L.03 new.ID := new.IDs[IDv]
L.04 UPDATE.IDS(IDv, old.IDs, new.IDs)
/* send the new labels to all other nodes */
L.05 SEND RELBL(IDv, old.IDs, new.IDs) TO all nb ∈ N_v
/* node v is done with its relabeling */
L.06 relabel := false

Macro REFRESH
R.01 Receiv.V, new.L, new.I := Ø
R.02 newlabels := false
R.03 ni := 0
/* node v warns the other pivots that it will start its relabeling */
R.04 for all id ∈ cover ∧ id ≠ IDv do
R.05 SEND CLEAN(IDv, id) TO H[id].neighbor
R.06 endfor

Macro RELABEL_DONE
/* check whether we have a new node or arc labeling function */
D.01 for all u ∈ V do
D.02 if (L'[u] ≠ new.L[u])
D.03 then
D.04 newlabels := true
D.05 endif
D.06 for all nbr ∈ N_v do
D.07 if ([IDv, nbr] ≠ new.I[u, nbr])
D.08 then
D.09 newlabels := true
D.10 endif
D.11 if (newlabels = true)
D.12 then /* we replace the old functions by the new ones */
D.13 for all u ∈ V do
D.15 endfor
D.16 for all nbr ∈ N_v do
D.17 I([IDv, nbr] := new.I[u, nbr]
D.18 endfor
D.19 newlabels := false
D.20 endif
/* we start a new phase */
D.21 RESTART

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.2.15 Pivot_Label

Macro RESTART;
T.01 exist.leader.sent.ball.cover.ready.relabel := false
T.02 reset2.reset3.reset4 := true
T.03 cover.S := Ø
T.04 updated := true
    /* eliminate from H all the nodes except B nodes */
T.05 for all id ∈ H \ B do
T.06 DeleteCell(id, H)
T.07 endfor

Macro UPDATE.IDS(current.id, pivot.id, oldIDs, newIDs)
Local.variables: u.nbr : int
U.01 Rcvd := Rcvd ∪ {pivot.id}
    /* we build the node labeling function */
U.02 for all u ∈ oldIDs do
U.03 new.L[u] := newIDs[u]
U.04 endfor
    /* we label the links toward a pivot p with the interval corresponding to its clients in S_{pivot.id} */
U.05 if (current.id ≠ pivot.id)
U.06 then
U.07 nbr := H[pivot.id].neighbor
U.09 endif
    /* now we label individual nodes from B and S */
U.10 for all u ∈ oldIDs ∧ u ≠ pivot.id ∧ u ∈ H do
U.11 nbr := H[u].neighbor
U.13 endfor

2. Clients action

When a non-pivot node receives a RELBL message:

- check if it is from a pivot and it is the first message received from that pivot.

- if yes, update all its variables and broadcast the message to the other neighbors.

Its new ID is kept in the variable newID and the value is used in the delivery protocol, until a new topology change occurs and we have to reset again.
Algorithm 4.2.16 Client Label

Messages: RELBL: sender: the ID of the sender
old.IDs: the old IDs of the nodes in $S_{sender}$ which have received a new label
new.IDs: the new labels of the nodes in $S_{sender}$
CLEAN: sender: the ID of the sender
dest: the ID of the destination

Local variables: Rcvd: set of int
$V$: set of int /* the set of current nodes in the graph */
nbr, nb, newID: int
newL: function from $V$ to $[1..n]$ /* new calculated node labeling function */
newJ: function from $E$ to $2^{|V|}$ /* new calculated arc labeling function */
newlabels: boolean /* is true if at least one label is changed */

Predicate: update.all $\equiv$ relabel $\land ID_v \notin$ cover

Macro: UPDATE.IDS $=$ build the interval routing scheme in the current node
input: id, pid: int
old.IDs, new.IDs: array of int

Actions:

s.01 relabel $\land (\text{pivot} \notin$ cover $\lor H \neq (B \cup$ cover $\cup S))$ $\rightarrow$ RESTART

s.02 update.all $\land (\text{resetA} = \text{true} \lor (\text{Rcvd} \setminus \text{cover} \neq \emptyset))$ $\rightarrow$

s.03 Rcvd, V, newL, newJ := 0
s.02 newlabels := false
s.04 resetA := false

s.05 Upon RECEIPT of CLEAN(s, dest) FROM nbr $\rightarrow$

s.06 if ($s \neq ID_v \land \text{dest} = ID_v$)
    then /* a message for me */
        Rcvd := Rcvd \ {s}
        /* node v warns its clients also */
        for all id $\in S \land id \neq ID_v$
            SEND CLEAN(ID_v, id) TO H[id].neighbor
    endfor

s.12 endif

s.13 if (choose.pivot $\land s \neq ID_v \land \text{dest} \in$ cover $\land \text{dest} \in H$)
    then /* a message for a pivot */
        SEND CLEAN(s, dest) TO H[dest].neighbor
    endif

s.16 Upon RECEIPT of RELBL(s, old.IDs, new.IDs) FROM nbr $\rightarrow$

s.16 if (update.all $\land s \in$ (cover $\setminus$ Rcvd))
    then
        Rcvd := Rcvd $\cup$ {s}
        V := V $\cup$ {old.IDs}
        if ($ID_v \in$ old.IDs)
            then /* the message contains our new ID */
                newID := new.IDs[ID_v]
            endif
            /* build the IRS using H and new IDs */
            UPDATE.IDS(ID_v, s, old.IDs, new.IDs)
            /* broadcast to the other neighbors */
            SEND RELBL(s, old.IDs, new.IDs) TO nb $\in V \setminus$ {nbr})
    endif

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
4.3 One example

Consider the network given as example in chapter 3, Figure 3.1. Because the system is asynchronous, we cannot bound the transmission time. Messages coming from the closest nodes are not always arriving first. For example, suppose that node 1 has received messages from its direct neighbors 7, 13, 10, 12 with the distances equal to the length of the direct links, and from node 8 with the distance \( \text{distance} = \text{length}(\text{link}(8,12)) + \text{length}(\text{link}(12,1)) = 5 \). Ordering the nodes in \( B_1 \) we have:

\[
\begin{array}{cccccc}
\text{node} & 12 & 7 & 8 & 13 & 10 \\
\text{distance} & 2 & 3 & 5 & 7 & 8 \\
\end{array}
\]

But this partial t-ball \( B_1 \neq B_1(5) = \{12, 7, 5, 8, 10\} \). So, node 1 has to wait until it will receive all the messages such that its final t-ball doesn’t change anymore, and it is finally:

Figure 4.1: A partial t-ball for node 1
Next, we prove the correctness of the algorithm.
CHAPTER 5

PROOF OF CORRECTNESS

The construction of the PIR starts with each node \( v \) calculating its partial t-ball \( B_v \). Based on all t-balls, the cover is calculated and therefore the nodes are divided into pivots and non-pivots. A pivot selects its client-hood (so the network is divided into \( k = |\text{cover}| \) client-hoods) and relabels the nodes inside its client-hood.

All the proofs are made for a generic node \( v \). First, we show that the partial t-ball \( B \) will contain only correct IDs. Based on correct t-balls, the calculated cover is a correct one. Second, that each non-pivot will elect the nearest pivot so SPIR builds in each pivot the SPT as \( PIR \). Based on SPTs, pivots do relabeling, so the IRS is correct and the same as in \( PIR \) scheme described in [EGP98].

For updating \( B \), \( v \) receives only correct distances. Besides the removal of the wrong IDs from \( B \) (Properties 5.1.7 and 5.1.8), we have to show that \( B \) gets emptied at most once (by executing \textit{RESTART}) in every execution of the algorithm (Lemma 5.1), so \( v \) can reach the last layer to build the labeling functions.

Once the cover is calculated, a non-pivot node \( u \) has to select its nearest pivot based on the distance values, so by Lemma 5.2 the nearer pivot becomes in finite time the nearest pivot for \( u \). Also, each pivot, once it knows all its clients, has to do the relabeling. By doing this in \textit{fair coordinated} manner (Lemma 5.4), we obtain correct labeling functions \( R = (L.I) \).

Each time the labeling functions are calculated, the old and the new values are compared. If there is at least one change, the old ones are replaced by the new ones. In any case, the algorithm restarts: the t-ball calculated up to that point is kept, but all the nodes except
the ones from \( B \) are erased from \( H \) and \( \text{cover} \). \( S \) are reset to \( \emptyset \). Only the guards of layer 1 and 2 are enabled, by setting the appropriate boolean variables to \( \text{true} \) or \( \text{false} \) value. So, it is a restart with \( \text{good} \) values. Later, new labeling functions are calculated and compared with the old ones, and we restart again. When we have no more changes in the \( t \)-balls in \( \text{cover} \) and all the \( S_p, p \in \text{cover} \), then the relabeling functions will remain the same.

Using Lemmas 5.5. 5.6. 5.7. 5.8, we prove that the algorithm stabilizes in \( O(d \sqrt{n(1 + \log n)}) \) time units, where \( n \) is the number of nodes and \( d \) is the diameter of the network. Therefore \( \text{SPIR} \) constructs a \( \text{PIR} \) scheme in polynomial time and it is self-stabilizing also.

5.1 A Correct T-ball

To calculate the \( t \)-ball \( B \) for an arbitrary node \( v \) in the network, we use the guarded commands in the algorithm \( \text{Calculate Ball} \) and some guards in \( \text{Error Correction} \).

Adding nodes to \( B \) is done automatically, and the macro \( \text{UPDATE} \) in \( \text{Calculate Ball} \) takes care of it. The main concern is to remove the “bad” nodes, with invalid information in \( H \) and/or \( B \). First, we show that the partial \( t \)-ball \( B \) will contain only correct IDs and the wrong IDs from \( B \) are removed (Properties 5.1.7 and 5.1.8.) Next, we prove that \( B \) can become \( \emptyset \) at most once, so \( B \) converges to a correct \( t \)-ball (Lemma 5.1.)

We have the following observations, most of them referring to macro \( \text{UPDATE} \) in \( \text{Calculate Ball} \):

**Observation 5.1.1** For \( \forall u \in B_v \), \( H_v[u].\text{distance} \) maintains in \( v \) the lowest distance received from \( u \) (lines U.10-13.) Starting from an arbitrary configuration, after a finite time \( H_v[u].\text{distance} \) is the lowest distance between \( v \) and \( u \).

Consider \( m \) as the initial value of \( H_v[u].\text{distance} \). If \( m \) is greater or equal to some distance from \( u \) to \( v \) received in some message, \( m \) gets later overwritten by that distance and \( H_v[u].\text{distance} \) converges to the shortest distance (condition \( H[id].\text{distance} > dist \) in line C.08 where \( id = u \) and \( dist \) is the value of the distance received in a message.)

If \( m \) is smaller than any possible value of the distance, we show that \( m \) gets replaced with a correct value and later \( H_v[u].\text{distance} \) converges to the shortest distance.
The value \( m \) is stored as the distance from \( u \) to \( v \) through a neighbor \( H_v[u].\text{neighbor} \). This neighbor keeps also \( u \) in its t-ball. otherwise it would not have forwarded the message. So, if that neighbor forwards to \( v \) another distance to \( u \), this value replaces \( m \). This action does not affect the process of selecting the shortest distance to \( u \) because the overwritten is done only in case a new distance is received from the neighbor toward \( u \) on the shortest path known up to this point. For example:

![Figure 5.1: A correct distance replaces the old one](image)

Assume now that \( v \) knows that \( u \) is at the distance 10 and this is the shortest distance to \( u \), so it is stored in \( H_v: H_v[u].\text{distance} = 10 \)

and \( v \) receives the distance from \( u \) through the path stored as shortest up to this point in \( H_v \) as 17. Then \( v \) replaces 10 by 17 : \( H_v[u].\text{distance} = 17 \)

and, maybe later, 17 will be overwritten by a shorter distance.

**Observation 5.1.2** \( B \) must contain the first \( t \) nodes in ascending order of the distance. If \( B \) has less than \( t \) elements, an incoming node is simply added to \( B \) (lines 18-19.) If \( B \) has exactly \( t \) elements and a new node should be added, the one with the longest distance, breaking ties by increasing node ID, is removed (lines 21-26.)

Another situation is the following. When we say that \( u \) is down, we see this from the point of view of \( v \); either \( u \) fails, or on the path from \( u \) to \( v \) some link is down, so \( v \) does not “see” \( u \) as an up node. If it is only a link failure, \( v \) will probably “see” \( u \) through another path.

For a node \( u \), \( u \neq v \), we have the following observations:
Observation 5.1.3 If u is stored in $H_v$ as a neighbor of v, but it is not a current neighbor ($u \not\in N_v$), u gets removed from $H_v$ (the guard 2.13 becomes true in node v and it is executed for id = u.)

Observation 5.1.4 If u is stored in $H_v$ as reaching v through a non-existing neighbor, u gets removed also (the guard 2.13 becomes true.)

Observation 5.1.5 If u reached v through a neighbor w of v ($w \in N_v$):

![Diagram](http://example.com/diagram.png)

Figure 5.2: The node u reached v through w but the path $u \rightarrow w$ is disconnected and the path between u and w does not exist anymore (we received a message LOST($w, u$)). then the path between u and v is removed from $H_v$ also (lines 2.15-19.)

Observation 5.1.6 From [EGP98] we know that:

If $u \in B_v$ ⇒ for all nodes w on the shortest path from u to v, $u \in B_w$.

Thus v checks this property with its direct neighbors by receiving their t-balls and sending its t-ball. whenever a change occurs in B.

Based on these observations, we prove further properties. The first property shows the removal of non-existing nodes. A non-existing node is a node which either has failed or it was never an up node in the network.

Property 5.1.7 (Removing the non-existing nodes) Starting from an arbitrary configuration, if a node u fails, or some links are down, or u does not exist, all nodes v. which have u included in $B_v$ as reachable through those links, will remove u from their B and H data structure in finite time.

Proof. The proof is by induction on the number of hops from u to an arbitrary v, such that $u \in B_v$. The general idea is.
a) if $u$ is stored as a direct neighbor of $v$ ($H_v[u].direct = true$):

\[ \text{Figure 5.3: A crash in the direct neighborhood} \]

then by Observation 5.1.3. $u$ is removed and the information is broadcast to the other nodes, as we remarked in Observation 5.1.6.

b) if $H_v[u].direct = false$ but $H_v[u].neighbor = u$, by Observation 5.1.3 or 5.1.4, $u$ is removed and the information is broadcast to the other nodes.

c) \( \exists w : H_v[u].direct = false \land H_v[u].neighbor = w \land w \neq u \). It is compulsory for $w$ to be an up neighbor, otherwise the guard 2.13 becomes true in node $v$ for $id = u$, and $w$ and all the other nodes which reach $v$ through $w$ get removed (Observation 5.1.4.)

We have a situation like this:

\[ \text{Figure 5.4: The down node } u \text{ reached } v \text{ through } w \]

$\exists w_k$ such that $u \in N_{w_k} \land H_{w_k}[u].direct = true \Rightarrow B_{w_k}$ and $H_{w_k}$ get updated. Recursively. $B_w$ and $H_w$ get updated. \( \square \)

**Property 5.1.8 (Removing the cycles)** Starting from an arbitrary configuration, for all the nodes in the network, the cycles in forwarding a message to any arbitrary node are eventually removed from $B$ and $H$.

**Proof.** A cycle in this case means that a node forward a message to another node, that node to another one so on, until the message is forwarded back to the first node.
without reaching the destination. For example, a cycle of dimension 3 can be:

![Figure 5.5: A cycle in delivering of dimension 3](image)

and we have the following values:

- $H_u[x].neighbor = w$ : $v$ knows that the best neighbor to reach $x$ is $w$
- $H_w[x].neighbor = u$ : $w$ knows that the best neighbor to reach $x$ is $u$
- $H_u[x].neighbor = v$ : $u$ knows that the best neighbor to reach $x$ is $v$

Therefore a message sent to $x$, once it enters the cycle, it goes forever. Simply checking whether $x \notin B_u \land H_u[x].neighbor = v \Rightarrow x \in B_v$ leave a cycle undetected.

A strong condition should be added such that, at some point, $x$ is removed from a set $B$ of a node along the cycle and recursively, $x$ gets removed from all the other nodes which form the cycle. The extra condition is $H_v[x].distance = length(v, w) + H_u[x].distance$ and it removes the eventual cycles.

To understand how it works, consider the following example:

![Figure 5.6: A correct situation](image)

Here, we see that $H_u[x].distance$ must be $2 + 3 = 5$.

How does this condition help us? Going back to Figure 5.6, suppose $H_v[x].distance = 12$:

- $H_u[x].distance$ must be $12 - 4 = 8$; $H_u[x].distance$ must be $8 - 3 = 5$; $H_u[x].distance$ must be $5 - 2 = 3$.

This is a contradiction to the initial value of $H_v[x].distance = 12$. □
Up to this point we have shown how to remove "bad" nodes from $B$. Another way to remove elements from $B$ is to set it to $\emptyset$. We show that it is possible at most once in every execution of the algorithm SPIR in node $v$. In this way, $B$ will contain, in finite time, the $t$ closest nodes to $v$. So gradually, the cover will be calculated based on these values, and finally the labeling functions.

Lemma 5.1 (Emptying $B$) Starting from an arbitrary configuration, in any execution, $v$ executes RESTART at most once.

Proof. Suppose we have executed RESTART and we analyze now what can happen next. We prove that further actions do not determine another RESTART. So, we analyze each guard from all the modules that have as action RESTART and we prove that they cannot become enabled again.

Property 5.1.9 After RESTART is executed, in the algorithm Error.Correction the predicate error remains false (so, its action RESTART is not executed anymore.)

Proof. After RESTART is executed, the data structure $B$ and $H$ are $\emptyset$. In the algorithm Calculate.Ball, whenever $B$ adds or removes an element, the same element is added/removed from $H$. Also, whenever a crash occurs in the network, the node $v$ does not become disconnected, so it has at least one up neighbor. So $B$ does not become $\emptyset$ because of Remove.ID executions. When RESTART is executed, we make sure to keep in $H$ whatever is in $B$. So, the condition $\langle B \setminus H \neq \emptyset \rangle$ is false from here on.

The statements in RESTART are executed atomically so, when $B$ is $\emptyset$, $H$, cover and $S$ are also $\emptyset$. They will be calculated later, based on the values of all the partial t-balls $B$ of all the nodes in the graph. Therefore, the condition $\langle B = \emptyset \land (\text{cover} \cup S) \neq \emptyset \rangle$ is false also from here on.

$S$ and cover start as $\emptyset$ and, as long as a cover is not calculated, cover remains $\emptyset$. Whenever cover is set to $\emptyset$, the variable cover.ready is set to false, so the guarded commands of the algorithms To.Pivots and To.Clients cannot execute in order to determine the set $S$ of the successors for the node. So $S$ remains $\emptyset$, therefore the condition $\langle \text{cover} = \emptyset \land S \neq \emptyset \rangle$ is false.
B. H. cover. S start as \( \emptyset \). Whenever \( B \) adds or removes an element, this element is added/removed from \( H \) in one atomic step. After the cover is calculated, the pivots send messages to all nodes and when such a message reaches the node \( v \), their IDs are added to \( H \). So \( H \) will eventually contain all the IDs stored in \( \text{cover} \). Also, whenever an ID is added to \( S \), it is added to \( H \) also. Therefore the condition \( H \nsubseteq (B \cup \text{cover} \cup S) \) is false.

When macro \textit{RESTART} is executed, the variable \( \text{to.pivot} \) starts with the ID of some neighbor (line R.04.) It can be changed in the following situations:

(i) the neighbor with the ID stored in \( \text{to.pivot} \) crashes (or the link to that neighbor crashes.) In this case, in the macro \textit{REMOVE} another neighbor is chosen (line R.05), so \( \text{to.pivot} \in (N_c \cup \{ID_c\}) \) remains true.

(iii) the node \( v \) is selected as a pivot, so \( \text{to.pivot} \) has (and keeps) the value \( ID_v \), so the condition \( \text{to.pivot} \in (N_c \cup \{ID_c\}) \) remains true.

(iv) the node \( v \) is not selected as a pivot, so it has to choose its nearest pivot. The variable \( \text{min.distance} \) keeps the lowest distance received toward any of the pivots. So whenever a lower distance toward a pivot is received, \( \text{to.pivot} \) is changed to that neighbor which sent this information. Therefore \( \text{to.pivot} \in (N_c \cup \{ID_c\}) \) remains true.

Property 5.1.10 Once the macro \textit{RESTART} is executed, the guard 1.04 of the algorithm \textit{ErrorCorrection} (keeping correct data about the neighbors) remains false.

Proof. The field \( \text{direct} \) specified whether a direct neighbor of \( v \) has the shortest path to \( v \) through the link between them.

\( H \) starts as \( \emptyset \). Whenever a node is added/updated in \( H \), the value of the field \( \text{direct} \) of that element is set to \( false \) (lines U.12, U.18, U.25. of the macro \textit{UPDATE} in the algorithm \textit{Calculate.Ball}, lines U.03, U.11 of the macro \textit{Update.H} in the algorithm \textit{ToClients}.)

The only statement which sets the value of the field \( \text{direct} \) for a node \( u \) to \( true \) is in the lines 1.06-1.08 of the algorithm \textit{ErrorCorrection}. But this is done only if \( u \) is neighbor of \( v \) and has the direct link as the shortest path \((H[u].\text{neighbor} = u)\) and, in this case the field \( \text{distance} \) is set to the correct value (the value of the length of the link.)

The field \( \text{distance} \) can change when a shorter distance is detected. But at that time.
the field \textit{direct} is set to \textit{false} automatically (lines U.12 of the macro \textit{UPDATE} in the algorithm \textit{Calculate.Ball}, line U.11 of the macro \textit{Update.H} in the algorithm \textit{To.Clients}.)

Taking an example:

![Figure 5.7: A shorter distance through another neighbor](image)

Suppose that initially for the node \( w \) stored in \( H_v \), the values of the fields are:

\[
H[w].\text{neighbor} = w \quad H[w].\text{distance} = 5 \quad H[w].\text{direct} = \text{true}
\]

If node \( v \) detects a shorter distance to node \( w \) through node \( u \), the values are changed in the macros \textit{UPDATE} (lines U.10-12) or \textit{Update.H} (lines U.09-11) to:

\[
H[w].\text{neighbor} = u \quad H[w].\text{distance} = 3 \quad H[w].\text{direct} = \text{false}
\]

\[\Box\]

\textbf{Property 5.1.11} The guard 2.02 in the algorithm \textit{Calculate.Ball} (keeping data about unknown nodes, whose \textit{DIST} messages have not been yet received) remains \textit{false}.

\textbf{Proof.} The set \textit{Rcvd.IDs} keeps all the nodes which have sent information regarding the distance toward the node \( v \). These distances are valid information.

Obviously, we cannot trust the information regarding a node in \( B \) whom message has not been yet received. Because of that we impose \textit{RESTART} when such a node is detected. After \textit{RESTART} gets executed and \( B \) becomes \( \emptyset \), \textit{Rcvd.IDs} := \{ID_v\}. From here on, \textit{Rcvd.IDs} will start storing only the IDs of the nodes which have sent messages and maybe have changed the set \( B \). So, \( B \subseteq \textit{Rcvd.ID} \), so the guard 2.02 remains \textit{false}. \[\Box\]

After \textit{RESTART} is executed, the variable \textit{pivot} is set to the node ID. If the node is part of the \textit{cover}, we make sure that \textit{pivot} and \textit{to.pivot} is the node ID. But if the node is not part of the cover, \textit{pivot} is set arbitrarily to one of the pivots (lines 6.10-12 in the algorithm \textit{To.Pivots}.) Starting from this point, \textit{pivot} is modified to the closest pivot whose
message has been received (macro \texttt{BE\lowercase{TTER}.PIVOT}.)

**Property 5.1.12** If \(v\) is a pivot, in the algorithm \texttt{Pivot.Label} the guard 7.01 (the chosen pivot is not in cover set or the data in the routing table are inconsistent) remains false.

**Proof.** The variable \texttt{pivot} is set to its own ID (line 5.06 in the algorithm \texttt{To.Clients}), so \(\texttt{pivot} \in \texttt{cover}\). The algorithm \texttt{Pivot.Label} runs only for the pivots. So, when \texttt{relabel} becomes \texttt{true}, this means that the node \(v\) has received messages from all other pivots and all non-pivot nodes. Besides the information regarding the nodes in \(B, H\) contains information about all other pivots and the successors (which are in fact its clients) and nothing else. Therefore \(H = (B \cup \texttt{cover} \cup S)\). \(\square\)

**Property 5.1.13** If \(v\) is not a pivot, in the algorithm \texttt{Client.Label}, the guard 8.01 (the chosen pivot is not in cover set or the data in the routing table are inconsistent) remains false.

**Proof.** The algorithm \texttt{Client.Label} runs only for the non-pivot nodes (a client of some pivot.) Thus, when \texttt{relabel} becomes \texttt{true}, this means that \(v\) has received messages from all the pivots and successors, and \(H\) contains this information. So, \texttt{pivot} is set to the nearer pivot (macro \texttt{BE\lowercase{TTER}.PIVOT} in the algorithm \texttt{To.Pivots}.) Later, the chosen pivot is checked if it is also part of the t-ball \(B_v\) (line 6.19 in the algorithm \texttt{To.Pivots}.) So, the condition \(\texttt{relabel} \land (\texttt{pivot} \notin \texttt{cover} \lor H \neq (B \cup \texttt{cover} \cup S))\) is false. \(\square\)

We have shown that all the possible guards which can determine a re-execution of \texttt{RE\lowercase{START}} remains \texttt{false} after a \texttt{RE\lowercase{START}} has been executed, so we have at most one \texttt{RE\lowercase{START}} in every execution. \(\square\)

So:

**Theorem 5.1.14 (Correct t-ball)** Starting from an arbitrary configuration, the partial t-ball \(B_v\), in finite time, becomes the -ball of node \(v\), \(B_v(t)\), required by \texttt{PIT} scheme.

**Proof.** Using the Properties 5.1.7, 5.1.8, Lemma 5.1. and Observation 5.1.1.
We know, by Property 5.1.7, that $B$ will contain only the up nodes in the network. and with the cycles removed (Property 5.1.8.) The guards of the algorithm Calculate_Ball updates $B$ in case of new distances or topology changes (Observation 5.1.1.)

By Lemma 5.1, once a node starts executing the distributed algorithm, we can have at most one \textit{RESTART}, which means that $B$ can be reset to $\emptyset$ at most once. □

5.2 Choosing the Nearest Pivot

When the set $B$ is changed and we supposedly have received at least one message from each other node, we go to the second level. The set $B$ containing the current t-ball is sent to the leader, and the variable \textit{ball.sent} is set to \textit{true} (line 2.25.)

A node, even if it is not done calculating its t-ball, forwards eventually t-balls received toward the \textit{leader} (line 3.03.) The leader collects the t-balls and once its t-ball is calculated ($\textit{ball.sent = true}$) and all other t-balls are received, it calculates a cover based on these t-balls, using a \textit{greedy} method. Whenever a new t-ball is received, \textit{leader} checks whether that node did not send one before. If the node has sent an old t-ball, the \textit{leader} replaces it with the new one (macro \textit{UPDATE.TBALLS}) and possibly recalculates again the cover. Once it is calculated, the cover is broadcast in the network and we go to the third level.

If $v$ is not selected as a leader, it forwards other eventual t-balls to the pivot. When the cover is received, the second level is done and we go to the third level.

When the algorithms in the second level are done, the variable \textit{cover.ready} becomes \textit{true} (lines 3.12, 4.15) for the third level.

In the third level, the nodes are divided into \textit{pivots} and \textit{clients}. A client selects a pivot as its nearest pivot. Each pivot arranges its clients in a SPT tree rooted at itself. Both have to maintain shortest distances to the all the pivots and the successors in the SPT tree which is part. (For a pivot, the successors are in fact all its clients.)

The criterion to choose a closer pivot is by distance. The next property proves that, regardless of the initial value of the variable \textit{min.distance} a closer pivot is chosen and in a finite time, the closest pivot will be eventually chosen. Once a non-pivot selects the correct pivot, it sends an acceptance message, so the pivots will have correct clients. So, it is enough
to prove that a client will eventually choose the nearest pivot in order to maintain that each pivot has correct clients. Also, the set of successors $S$ maintained in each node depends of the correct selection of the pivots by the non-pivots.

**Lemma 5.2 (min\_distance)** Starting from an arbitrary configuration and regardless of its initial value, the variable min\_distance in the algorithm To\_Pivots will have in a finite time, the minimum value among all the distances toward the pivots (the nodes in cover.)

**Proof.** Generally, when we receive a message from a pivot, we compare min\_distance with the distance received, stored in the variable dist. If min\_distance > dist, we update min\_distance and eventually it will converge to the minimum distance among all the pivots.

But if we start with a small value for min\_distance, such that always min\_distance < dist, we will not be able to converge to the correct pivot.

In order to correct this, suppose that $p$ is the starting pivot node for $v$, $p \in$ cover, and nbr is the starting neighbor toward it. Thus, min\_distance is what the node $v$ knows that is its distance to $p$ through the neighbor nbr.

When we receive a message from $p$ through nbr containing the valid distance, we simply modify min\_distance to that distance. Possibly this real distance is greater than some distances already received and discarded. But we know that a node is continuously broadcasting its distance toward the other node, so after a while this valid value of min\_distance can be properly changed.

We do not know when the message arrives but it will arrive, because each of the neighbors (pivots or not) maintains shorter distances toward all the pivots. Therefore we will receive messages regarding each pivot from all of them, so the erroneous value of min\_distance is eventually corrected.

Consider the following example. The node $v$ has the variable min\_distance = 3 and the current nearer pivot $u$, $u \in$ cover. The neighbor toward $u$ on the current shortest path is $w$. Now, node $v$ receives a message through $w$ with another distance toward $u$. We know that this is the actual value, because the message is not corrupted and it carries a correct value, and all other nodes on the path toward $u$ have done the change in their min\_distances.
Figure 5.8: Increasing \textit{min.distance}

Now, it is the turn of the node \( v \) to replace the old value of \textit{min.distance} which is 3 with the newer one. \( \Box \)

### 5.3 Relabeling

We already know that a non-pivot node \( v \) always chooses its pivot as a node from the \textit{cover}. But that pivot should be also in its t-ball: \( \textit{cover} \cap B_v \neq \emptyset \Rightarrow \exists p \in \textit{cover} \land p \in B_v \).

We wait until we have received at least one message from each pivot and the chosen pivot is in \( B \). At that time \( v \) considers the pivot elected and sends an acceptance message to that pivot and refusals to all the other pivots. And the third level ends until local or global changes requires a re-execution of this level. The condition \( \text{Revd} = \textit{cover} \) is necessary to be \textit{true} in order to proceed for choosing the closer pivot because we have to send refusals to all other pivots, so we need at least one path to each of them. That path will not be in that moment the shortest, but in a finite time it will converge to it.

For a pivot the third level ends when it receives acceptance or refusals from all nodes which are non-pivots and messages regarding the distance from all other pivots.

When the third level is done the variable \textit{cover.ready} becomes \textit{false} (lines A.05 in the macro \textit{ALL ANSWERS}, 6.25), until we need to execute the algorithms on this layer again, and we go to the fourth layer by setting the variables \textit{relabel} and \textit{reset4} to \textit{true} (lines A.04, 6.24.)

The purpose of our distributed algorithm is to calculate the labeling functions \( \mathcal{L} \) and \( \mathcal{I} \). All other layers concur for providing \textit{enough} information such that this step is done properly.
Each pivot $p$ relabels the node in its partition. Having the clients arranged in a SPT tree (called $S_p$) rooted at itself, the pivot starts re-numbering the nodes in a preorder traversal of the tree. This method, proposed by Santoro and Khatib [SK85], is the first compact routing method and is called tree-labeling. Once the process is done, $p$ sends messages to all the other nodes and constructs the portion of the functions which define the IRS.

The SPT is correctly constructed, by Lemma 5.2. So the most important aspect is the order in which the pivots start relabeling their clients. In the Lemma 5.4 we prove that the order is the one required by [EGP98], so the pivots, running concurrently, label in fair coordinated. Next, we prove this property and the fact that one message accepted from each pivot is enough for all the nodes in the graph to eventually construct correct labeling functions (Lemma 5.3.) We restrict ourselves to the case of one message accepted, because of the asynchronous model. An old relabeling can reach a node later than a newer one, so it can overwrite it. In order to prevent this, we allow a node to accept only one message with relabeling, and that message should follow a CLEAN message, which specifies that a certain pivot will start the relabel. Because we assume that we do not have message reorder, a RELBL message will reach a node after a CLEAN message has already been received. And a node will receive (and accept) only one CLEAN message, sent or forwarded by its own pivot and not by anybody else. In this way, we prevent flooding the network with confusing CLEAN or RELBL messages.

**Lemma 5.3 (One message per pivot)** For a pivot node $v$, regardless of the initial value of the Rcvd, accepting only one message from each other pivot does not affect the process of building correct IR functions.

**Proof.** The condition 7.10 in the algorithm PivotLabel imposes the restriction of at most one message from each pivot. We can have no more messages accepted from a pivot $p$ if $p \in \text{Rcvd}$ because of a possibly wrong initialization and not because a message from $p$ has been received. Here we have several cases:

(i) the client-hood of $p$ interferes with the client-hood of $v$. In this case, some guards in $To.Pivots$ become true and a new relabeling starts for $v$ with relabel, reset4 := true.
In this case, the guard 7.02 becomes true for both \( v \) and \( p \). \( Rcvd := 0 \) for \( v \), so a new message from \( p \) is awaited.

(ii) the client-hood of \( p \) does not interfere with the client-hood of \( v \), but interfere with the client-hood of another pivot \( q \). then both \( p \) and \( q \) will have a new relabeling so appropriate \textit{CLEAN} messages will remove them from \( Rcvd \) of \( v \) such that the new relabels of \( p \) and \( q \) will be accepted by \( v \).

(iii) the client-hood of \( p \) does not interfere with any client-hood. it is a simply re-arrangement of the clients. The node \( p \) will send a \textit{CLEAN} message to determine the node \( v \) to remove \( p \) from \( Rcvd \), so the new relabels of \( p \) will be accepted by \( v \).

\( \square \)

A node accepts only one message \textit{RELBL} is order to prevent the network to be flooded.

For a pivot, we need to prove that the relabeling process respects the \textit{PTR} preprocessing step of [EGP98]. Or simply, the fair-coordination among the pivots dictates when each pivot does its relabeling.

\textbf{Lemma 5.4 (Fair-coordination)} The pivots relabeling is done in fair-coordinated manner.

\textbf{Proof.} We know that the first pivot from \textit{cover} is the one which starts the process of relabeling for the entire network. We order the cover = \{\( p_1, p_2, \ldots p_k \}\).

Each pivot starts in some state and once reaches the fourth layer, first it ensures that all the other nodes will accept its new value by sending a message \textit{CLEAN} first. Because we have FIFO channels when the message \textit{RELBL} containing the relabel reaches a node, we know that \textit{CLEAN} was previously received and the node is able to accept the relabels. \textit{CLEAN} does not require any condition to be true to be processed, because a different node can execute guarded commands of different layers with various speed of execution, so we cannot expect all the pivots to be in fourth layer at the same time.

The node \( p_1 \) starts relabeling first once \textit{relabel} = \textit{true}. After receiving \textit{CLEAN} the other pivots wait for the message of \( p_1 \). Once this is done, it broadcasts the new labels.
in the network to all the nodes. For a non-pivot node, only the \textit{CLEAN} messages are restricted to be sent be its pivot. \textit{RELBL} messages are accepted from any node, but only the first message. Once \(p_2\) receives the message from \(p_1\), it is its turn to do relabeling, and so on. \(\square\)

When a node is assumed to have received relabels from all the pivots, it checks whether the labeling functions are different from the previous ones. If there is at least one change, the old ones are replaced by the new ones. The algorithm restarts: The t-ball calculated up to this point is kept, but all the nodes except the ones from \(B\) are erased from \(H\). The variables \(\text{cover}, S\) are reset to \(\emptyset\). Only the guards of layer 1 and 2 are enabled, by setting the appropriate boolean variables to \textit{true} or \textit{false} value. Thus it is a restart with \textit{good} values. Later, new labeling functions are calculated and compared with the old ones, and we restart again. When we have no more changes in the t-balls, in \(\text{cover}\) and all the \(S_p, p \in \text{cover}\), the relabeling functions will remain the same.

5.4 Self-Stabilization and Time Complexity

We consider \(d\) to be the diameter of the network. In case of network change, \(d\) can be modified, so, to be more precise, consider \(d\) to be the maximum diameter over all the diameters of the network in different situations which have occurred (in the worst case \(d = n\).)

Also, \(S_p\) represents the set of clients for each pivot \(p \in \text{cover}\), calculated in the algorithm \text{To.Clients}.

Because the property of \textit{self-stabilization} implies that the system will reach a correct state in finite time, we prove the self-stabilization together with an analysis of the time.

We define the state predicate \(\mathcal{L}_I\), \(\mathcal{L}_I = I_1 \land I_2 \land I_3 \land I_4 \land I_5\) as the invariant for all legitimate states:

\(I_1\) : \(B_v = B_v(t), \forall v \in V\) - indicates that the set \(B\) calculated in each node is the defined t-ball \(B_v(t)\) (or, the partial t-ball is the t-ball for each node \(v\) in a legitimate state.)

\(I_2\) : \(\text{cover} = \text{Calculate}\_\text{cover}\{B_v(t)\} v \in V\) (the cover calculated by the algorithm...
Calculate.Cover is the cover for all t-balls.)

$I_3 : \text{pivot} = p_v, \forall v \in V - \text{cover} \text{ (the nearer pivot calculated in the algorithm To.Pivots is the nearest pivot in terms of the distance } \prec_v \text{, for each non-pivot node.)}$

$I_4 : V = \bigcup_{v \in \text{cover}} S_v \cup \text{cover and } S_p \cap S_i = \emptyset, \forall p.t \in \text{cover}.p \neq t.$

(the set of the clients \( \{S_v|v \in V\} \) forms a partition for the nodes in the graph)

$I_5 : (L, I) \text{ constructed in the macro UPDATE.IDS is the PIR scheme.}$

**Lemma 5.5** $I_1$ is a closed attractor for $C$.

**Proof.** By Theorem 5.1.14. starting from an arbitrary state, in finite time, for each node $v$ the set $B_v = B_v(t)$.

By Lemma 5.1. once a node starts executing the distributed algorithm, we can have at most one RESTART, which means that $B$ can be reset to $\emptyset$ at most once. So, if we have a RESTART consider $t_R$ the time spent until all RESTART gets executed. Time $t_R$ depends on the local computation.

And consider $t_1$ time units to received messages from all the other nodes. Time $t_1$ depends on the diameter of the network, because a message can come from at most distance $d$, and on the time spent by the processors on the path through which the message reaches $v$, to process and forwards the message to $v$. If no RESTART is executed in any node in the network, for each node $v \in V$, in $O(d)$ time units, all the distances are received and the partial t-ball $B_v$ becomes the t-ball $B_v(t)$. So, the set $B$ calculated by the algorithm Calculate.Ball contains the $t$ nearest nodes to $v$.

Therefore it will take at most $t_R + O(d)$ time units for $v$ to calculate its t-ball $B_v(t)$ in the variable $B$. □

**Lemma 5.6** $I_2$ is a closed attractor for $C$.

**Proof.** Consider leader the node elected as a leader.
If all the nodes have their t-balls calculated and no RESTART occurs anymore, in \( O(2d) \) time units the elected leader eventually collects all the t-balls: each node needs \( O(d) \) time units to calculate the t-ball and \( O(d) \) time units to send it to the leader.

By Lemma 5.5, a node needs a finite time to calculate its t-ball. Once it is computed, the t-ball is sent to the leader in the message BALL through the neighbor nbr. locally computed by each node. Consider the worst case, when the distance in hops between the elected leader and a node is \( d \). so BALL will reach the leader in \( O(d) \) time units.

After a node has received messages from all the other nodes in the network, it sends the set \( B \), but continues to run. Whenever a change occurs in \( B \) due to a smaller distance received, \( B \) is sent again to the leader. Within \( d \) time units, the node \( v \) has to receive all the messages sent by the other nodes, so it has to wait maximum \( d \) time units to have received messages from its \( t \) nearest nodes and to have the partial t-ball \( B \) to its t-ball.

Now, leader has to wait \( O(2d) \) time units (from all the \( n \) nodes) to receive the correct t-balls and, based on their values, to compute the cover. Once the cover is calculated, in \( O(d) \) time units each node will eventually receive it.

\[\square\]

**Lemma 5.7** \( I_3 \) and \( I_4 \) are closed attractors for \( C \).

**Proof.**

We have to prove that after the cover is received by any node, and no more RESTART is executed in any node:

(i) within \( d \) time units, for each non-pivot node, the nearer pivot becomes the nearest one

(ii) within \( d + d = 2d \) time units, each pivot has its set of clients selected

We know that the set \( B \) contains at least one pivot, and by Lemma 5.2 we know that each non-pivot node \( v \) will eventually choose its nearest pivot. We cannot say that this can be done in \( t \) time units, because for selecting nodes in \( B \) the distance is not considered in terms of hops, but in real values. So, each non-pivot has to wait at most \( d \) time units, to receive all the messages from other nodes.
Each pivot has to wait for a client to elect it as the nearest pivot and then to send the answer to it. Because a client has a distance in hops to the pivot of at most $d$, the answer sent by the client node will reach the pivot in at most $d + d = 2d$ time units, not taking in consideration the time spent by the client in local computation.

So, by Lemma 5.7 in finite time, each node is ready to start the fourth layer, which focuses on relabeling and calculating the functions for IRS.

**Lemma 5.8** $L_5$ is a closed attractor for $C$.

**Proof.** Consider the moment when $\text{relabel} = \text{true}$ for every node and $\text{RESTART}$ is not executed anymore in any node. Within $d$ time units, each node receives the labels sent by the first pivot $p_1$ (including the second pivot $p_2$.) Within $d + d = 2d$ all nodes receive the labels sent by the second pivot $p_2$.

Using Lemma 5.4, the relabeling is done as required by the PIR, and by accepting one message per pivot (Lemma 5.3.) So, in $O(kd)$, where $k = |\text{cover}|$, each node receives all the labels. It takes $O(kd)$ time units for the relabeling functions of IRS to be completely constructed in each node (macro $UPDATE. IDs$.) By [EGP98], $k = |\text{cover}| \leq \sqrt{n(1 + \log n)}$. so the stabilization time becomes $O(d\sqrt{n(1 + \log n)})$ time units.

**Theorem 5.4.1 (Closure and convergence of $L_I$)** The distributed algorithm constructs a PIR scheme in polynomial time. The characteristics of the PIR scheme are preserved, as in the [EGP98]: the stretch factor is at most five and is three on the average, and the routing table size of size $O(n^{1/2}\log^{3/2} n + \Delta v \log n)$ bits per node and with a total of $O(n^{3/2}\log^{3/2} n)$. The algorithm stabilizes in $O(d\sqrt{n(1 + \log n)})$ time units.

**Proof.** By transitivity, using Lemmas 5.5, 5.6, 5.7, and 5.8.
CHAPTER 6

CONCLUSIONS

In this thesis, we presented a self-stabilized interval routing scheme $SVTR$. As high speed networks become larger and larger, it is essential to design direct routing schemes, which require a relatively small amount of memory in the nodes for routing purposes. The proposed algorithm is the first self-stabilizing compact routing algorithm for an asynchronous, arbitrary weighted network, and can easily be extended to an unweighted network. It takes $O(d\sqrt{n(1+\log n)})$ time units to stabilize, where $n$ is the number of nodes and $d$ is the diameter of the network, and the routing functions use $O(n^{3/2}\log^{3/2} n)$ bits in total.

Interval routing algorithms can be used to design efficient solutions to some fundamental problems in distributed computing, such as broadcasting, mutual exclusion, BFS, and DFS. There already exist self-stabilizing solutions to the above problems [Dol00]. One interesting topic of future research is to find efficient self-stabilizing solutions (more efficient than the existing ones) to the above problems.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Doina Bein

Local Address:
2355 Brockton Way
Henderson, NV 89014

Home Address:
2355 Brockton Way
Henderson, NV 89014

Degrees:
Bachelor of Science. Computer Science. 1996
Al. I. Cuza University of Iasi. Romania

Al. I. Cuza University of Iasi. Romania

Thesis Title: Self-stabilizing Interval Routing Scheme in General Networks

Thesis Examination Committee:
Chairperson. Dr. Ajoy Kumar Datta. Ph.D.
Committee Member. Dr. Thomas Natker. Ph.D.
Committee Member. Dr. John T. Minor. Ph.D.
Graduate Faculty Representative. Dr. Henry Selvaraj, Ph.D.

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