1-1-2001

A self-stabilizing interval routing scheme in general networks

Doina Bein
University of Nevada, Las Vegas

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

UNLV

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

June 11, 2001

The Thesis prepared by

Doina 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

Ajoy Kumar Datta
Examination Committee Chair

[illegible]
Dean of the Graduate College

[illegible]
Examination Committee Member

John T. Minor
Examination Committee Member

[illegible]
Graduate College Faculty Representative

PR/1017-53/1-00

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 Δ_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

LIST OF FIGURES

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 $\Omega(n^{1+\frac{1}{2s+4}})$ 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 $\Omega(k^3 n^{1+1/k} \log n)$ 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 a 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 $\mathcal{PIR}$ 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. $\mathcal{PIR}$ 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* $\mathcal{SPIR}$. It is a fault-tolerant distributed implementation of the $\mathcal{PIR}$ 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 Δ_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 $\mathcal{PIR}$ 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 $\mathcal{PIR}$ 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 $\mathcal{PIR}$ 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 $\prec_p$. The number of neighbors of p. $|N_p|$. is called the *degree* of p and is denoted by Δ_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*:

$$< label >::< guard >\rightarrow< action >$$

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

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

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

The set of computations of $\mathcal{P}$ in S starting with a particular configuration $\alpha \in \mathcal{C}$ is denoted by $\mathcal{E}_\alpha$. A configuration β is *reachable* from α, $\alpha \rightsquigarrow \beta$, if there exists a computation $e \in \mathcal{E}_\alpha, e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots)$ 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 [Dol00]. 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 $\mathcal{L}_{\mathcal{P}}$ 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 $\mathcal{PIR}$ 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: *for any* $c \in C$, $c \vdash$ true.

We introduce the concept of an *attractor* to define self-stabilization. Intuitively, an attractor 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 $\mathcal{S}$ is called self-stabilizing if and only if there exists a non-empty subset $\mathcal{L} \subset \mathcal{C}$ of legitimate configurations and a predicate $\mathcal{L}_\mathcal{P}$ such that $\mathcal{L}$ is a closed attractor for $\mathcal{C}$:*

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

(ii) *true* $\triangleright \mathcal{L}_\mathcal{P}$ *(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 sends 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 in 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 $(\mathcal{L}, \mathcal{I})$, generated in the preprocessing step, where $\mathcal{L}$ is a node-labeling, $L : V \rightarrow \{1, \ldots, n\}$, and $\mathcal{I}$ is an arc-labeling, $I : E \rightarrow 2^{L(V)}$. 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 *Comp(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 $IRS(S) = 1$ 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 $\mathcal{I}(e) \subset \{1, 2, \ldots, n\}$ 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:

$$Stretch_G(R) = \max_{x \neq y} \frac{dist_G(R, x, y)}{dist_G(x, y)}$$

$$AvStr_G(R) = \frac{1}{n(n-1)} \sum_{x \neq y} \frac{dist_G(R, x, y)}{dist_G(x, y)}$$

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 $dist(x, v) < dist(y, v)$ or $dist(x, v) = dist(y, v)$ and $ID(x) < ID(y)$.

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 contains 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 $L(v) \in I(e)$.

a. Preliminary construction:

1. Let P be a cover for the collection of t-balls of the nodes, $\{B_v(t) | v \in V\}$.
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 $S_p = \{v | p(v) = p\}$ 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:

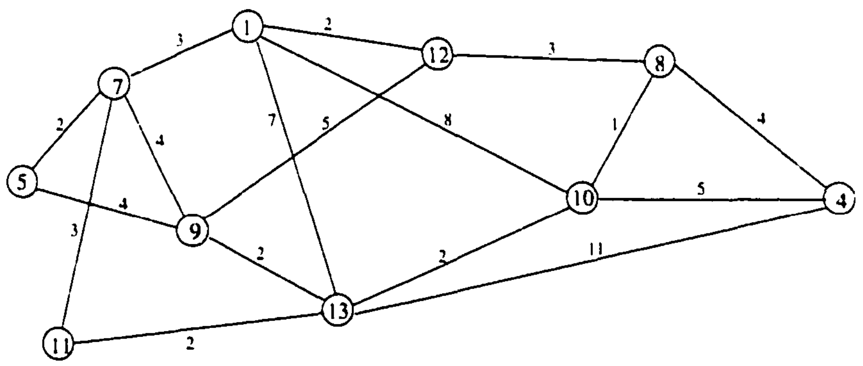

Figure 3.1: An arbitrary network

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

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

$B_v(t) = B_v(5). \forall v \in \{1.4.5.7.8.9.10.11.12.13\}$ (6 = 5 + 1 which is the current node. not included in its t-ball.)

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 $B_1(5)$ is :

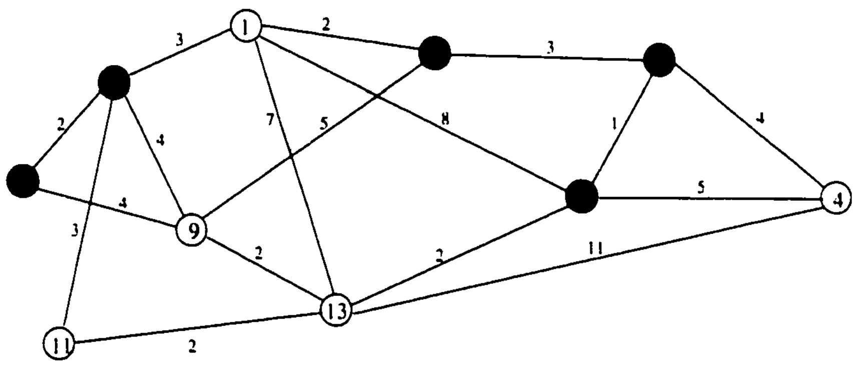

Figure 3.2: The t-ball $B_1(5)$

and the nodes from $B_1(5)$ are shadowed. We have:

$B_1(5)$					
node	12	7	5	8	10
distance	2	3	5	5	6

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:

$B_1(5) = \{12.7.5.8.10\}$

For other nodes:

$B_4(5)$							
node	8	10	12	13	1	9	11
distance	4	5	7	7	9	9	9

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.

$B_5(5)$

node	7	9	1	11	13
distance	2	4	5	5	6

$B_7(5)$

node	5	1	11	9	12	13
distance	2	3	3	4	5	5

$B_8(5)$

node	10	12	13	4	1	11
distance	1	3	3	4	5	5

$B_9(5)$

node	13	5	7	10	11
distance	2	4	4	4	4

$B_{10}(5)$

node	8	13	9	11	12
distance	1	2	4	4	4

$B_{11}(5)$

node	13	7	9	10	5	8
distance	2	3	4	4	5	5

$B_{12}(5)$

node	1	8	10	7	9
distance	2	3	4	5	5

$B_{13}(5)$

node	9	10	11	8	7
distance	2	2	2	3	5

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 :

node	1	4	5	7	8	9	10	11	12	13
frequency	5	1	4	6	5	6	7	5	5	6

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 (B_5, B_7, B_{10}) are computed again. The new values are:

node	1	4	5	7	8	9	10	11	12	13
frequency	2	0	1	1	1	3	0	3	2	2

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:

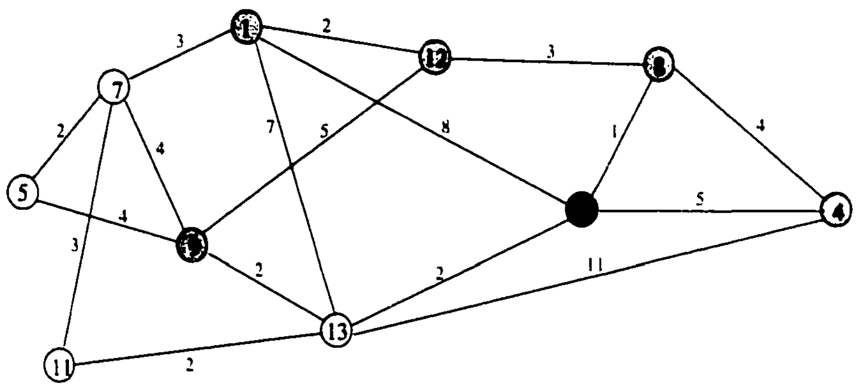

Figure 3.3: The pivots and their clients

- 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 $dist(11, 9) < dist(11, 10)$.
- node 12 has its *pivot* = 10 even if both $9, 10 \in B_{12}(5)$, because $dist(12, 10) < dist(12, 9)$.
- node 13 has its *pivot* = 9 even if both $9, 10 \in B_{13}(5)$, $dist(13, 9) = dist(13, 10)$, but $9 < 10$.

CHAPTER 4

THE DISTRIBUTED PIVOT INTERVAL ROUTING SCHEME

We present a fault-tolerant distributed implementation of the $\mathcal{PIR}$ 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 v maintains several variables of different types. One variable is computed by the underlying layer of topological maintenance protocol :

N_v = the set of the neighbors' IDs of the node v

leader = the elected leader for all the nodes in the graph

to_leader = 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 $\sqrt{n * \ln n}$

pivot contains the nearest pivot

to_pivot the neighbor which is the first node on the shortest path to that particular pivot

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

*reset*2, *reset*3, *reset*4 are true when we restart the algorithm for the layer 2, 3, 4

iii) other data structures :

L, l 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

RcvdIDs 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 $\emptyset$

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:

- $v \in H$ means $v \in Give\_IDs(H)$ (e.g. $H \not\subseteq (B \cup cover \cup S)$ means $Give\_IDs(H) \not\subseteq (B \cup cover \cup S)$)

- $H[id]$ 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*	- all nodes except the *leader* -
	or	
	Alg. *Calculate_Cover*	- the *leader* -
Third layer:	Alg. *To_Clients*	- the *pivots* from *cover* -
	or	
	Alg. *To_Pivots*	- the *non-pivots* -
Fourth layer:	Alg. *Pivot_Label*	- the *pivots* from *cover* -
	or	
	Alg. *Client_Label*	- the *non-pivots* -

The general algorithm is:

Algorithm 4.2.1 $\mathcal{SPIR}$ Stabilizing Pivot Interval Routing Scheme

```
A.01        Error_Correction
A.02        Calculate_Ball
A.03        Receiving_Cover
A.04        Calculate_Cover
A.05        To_Clients
A.06        To_Pivots
A.07        Pivot_Label
A.08        Client_Label
```

Error_Correction has the role to broadcast periodically a message $DIST$ 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 *Calculate_Ball* gathers information about the neighborhood. The $DIST$ 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_v 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 *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 *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 $\mathcal{SPIR}$ 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:  id_1: the sender ID
                 id_2: the ID of the other node adjacent to the crashed link
Local_variables: id, nb, nbr: int
Predicate error ≡ $(B \setminus H \neq \emptyset) \vee (B = \emptyset \wedge (H \cup cover \cup S) \neq \emptyset) \vee (cover = \emptyset \wedge S \neq \emptyset) \vee$
                  $\vee (H \,^* (B \cup cover \cup S)) \vee (to\_pivot \notin (\mathcal{N}_v \cup \{ID_v\}))$
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 $nb \in \mathcal{N}_v$

1.03  error  ⟶  RESTART

1.04  $\exists id \in H : H[id].direct = true \wedge id \in \mathcal{N}_v \wedge length(\text{link to } id) \neq H[id].distance$  ⟶
1.05      RESTART

1.06  $\exists id \in H : H[id].direct = false \wedge id \in \mathcal{N}_v \wedge H[id].neighbor = id$  ⟶
1.07      $H[id].distance := length(\text{link to id})$
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  $RcvdIDs := \{ID_v\}$
R.05  $to\_pivot :=$ some neighbor from $\mathcal{N}_v$
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 $N_v$ */
                  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 */
```

Algorithm 4.2.4 *Calculate_Ball*

```
        $RESTART_1$   = 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 $H$        input: $id, dist, nbr$: int
Actions:

$ID_v = leader$    $\longrightarrow$    $exist\_leader := true$

$(B \cup \{ID_v\}) \setminus Rcvd\_IDs \neq \emptyset$    $\longrightarrow$    $RESTART$

$\exists id \in H : (id \notin N_v \wedge H[id].direct = true) \vee H[id].neighbor \notin N_v$    $\longrightarrow$
    /* $id$ is a wrong node and remove it from $B$ and $H$ */
    $REMOVE(id, N_v)$ [illegible]
        $Rcvd\_IDs := \{ID_v\}$
        $H := \emptyset$
        $RESTART_1$

Upon RECEIPT of $DIST(s, dist_s)$ FROM neighbor $nbr$    $\longrightarrow$
    if $(s \neq ID_v)$
    then
        $dist_s := dist_s + length$(link to $nbr$)
        /* update the information in $B, H$ */
        $UPDATE(s, dist_s, nbr)$
    endif

Upon RECEIPT of $LOST(id_1, id_2)$ FROM $nbr$    $\longrightarrow$
    if $(id_2 \in H \wedge H[id_2].neighbor = id_1)$
    then
        $REMOVE(id_2, N_v \setminus \{nbr\})$
    endif

$(|Rcvd\_IDs| = n \wedge updated \wedge exist\_leader)$    $\longrightarrow$
    if $(ID_v \neq leader)$
    then
        SEND $BALL(ID_v, B, leader)$ TO to_leader
    endif
    $ball\_sent := true$
    $updated := false$

$(|Rcvd\_IDs| = n \wedge to\_send)$    $\longrightarrow$
    SEND $CHECK(ID_v, B, H)$ TO all $nb \in N_v$
    $to\_send := false$

Upon RECEIPT of $CHECK(s, B_s, H_s)$ FROM nbr    $\longrightarrow$
    if $(s = nbr)$
    then
        for all $(u \in B \wedge u \neq s \wedge H[u].neighbor = s)$ do
            if $(u \notin B_s) \vee (u \in B_s \wedge H[u].distance \neq H_s[u].distance + length$(link to $nbr$))
            then
                $REMOVE(u, N_v)$
            endif
        endfor
    endif
```

Algorithm 4.2.5 *Calculate_Ball*

```
Macro $REMOVE(id, N)$
R.01  $RESTART_1$
R.02  $Remove\_ID(id, H, B)$
R.03  if $(to\_pivot = id)$
R.04  then
R.05      $to\_pivot :=$ some neighbor from $N_v$
R.06  endif
R.07  SEND $LOST(ID_v, id)$ TO all $nb \in N$
R.08  $Rcvd\_IDs := Rcvd\_IDs \setminus \{id\}$
Macro $RESTART_1$
T.01  $ball\_sent, cover\_ready, relabel := false$
T.02  $reset2, reset3, reset4 := true$
T.03  $cover, S := \emptyset$
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 $nb \in N_v \setminus \{nbr\}$
S.02  $updated, to\_send := true$

Macro $UPDATE(id, dist, nbr)$
Local_variables:  $ID\_max\_dst$ : int

U.01  $Rcvd\_IDs := Rcvd\_IDs \cup \{s\}$
U.02  if $(id = leader)$
U.03      then
U.04          $exist\_leader = true$
U.05  endif
U.06  if $(id \in B)$
U.07  then
U.08      if $(id \in H \wedge (H[id].distance > dist \vee (H[id].distance < dist \wedge H[id].neighbor = nbr)))$
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 $(H[ID\_max\_dst].distance > dist) \vee H[ID\_max\_dst].distance = dist \wedge ID\_max\_dst > id))$
U.23          then /* $id$ is inserted and $ID\_max\_dst$ is removed */
U.24              $RESET_1(ID\_max\_distance, N_v)$
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 $v \neq$ *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*

```
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: id, dest, nbr : int /* elements in $N_v$ */
                 $B_s$, cov: set of $t$ int
Predicate: wait_cover ≡ ball_sent $\wedge$ $ID_v \neq$ leader
Macro: $RESTART_2$ = restart the layers 3 and 4
Actions:

3.01  wait_cover $\wedge$ reset2 = true  $\longrightarrow$  $RESTART_2$

3.02  Upon RECEIPT of $BALL(s, B_s, dest)$ FROM nbr  $\longrightarrow$
3.03      if ($ID_v \neq$ leader $\wedge$ $s \neq ID_v$ $\wedge$ nbr $\neq$ to_leader $\wedge$ dest = leader)
3.04      then /* a message to be forwarded to the leader */
3.05          SEND $BALL(s, B_s, dest)$ TO to_leader
3.06      endif

3.07  Upon RECEIPT of $COV(s, cov)$ FROM nbr  $\longrightarrow$
3.08      if (wait_cover $\wedge$ s = leader $\wedge$ nbr = to_leader $\wedge$ cover $\neq$ cov)
3.09      then /* a new cover from the leader */
3.10          cover := cov
              /* broadcast the cover to the other neighbors */
3.11          SEND $COV(s, cov)$ TO $nb \in N_v \setminus \{nbr, to\_leader\}$
3.12          cover_ready := true
3.13          $S := \emptyset$
3.14      endif

Macro $RESTART_2$
R.01  cover_ready, relabel := false
R.02  reset3, reset4 := true
R.03  cover, $S := \emptyset$
R.04  reset2 := 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: $determine\_cover \equiv ball\_sent \wedge ID_v = leader$
Macro: $UPDATE\_TBALLS$ = update the current set of tballs by adding new or replacing the old ones
                          input: $id$ : int
                                 $B_td$ : set of $t$ int
                                 $TBalls$ : Linked_List of (int, set of $t$ int)

Actions:

4.01  $determine\_cover \wedge reset2 = true$   $\longrightarrow$
          /* eliminate all the t-balls received */
4.02      $update := false$
4.03      $Rcvd, TBalls := \emptyset$
4.04      $RESTART_2$

4.05  Upon RECEIPT of $BALL(s, B_s, dest)$ FROM $nbr$   $\longrightarrow$
4.06      if $(ID_v = leader \wedge dest = ID_v)$
4.07      then
4.08          $UPDATE\_TBALLS(s, B_s, TBalls)$
4.09      endif

4.10  $determine\_cover \wedge |Rcvd| = n - 1 \wedge update$   $\longrightarrow$
          /* we can calculate the cover for all the t-balls including the local one $B^*$ */
4.11      $UPDATE\_TBALLS(ID_v, B, TBalls)$
4.12      $cover := Greedy\_Cover(TBalls)$
          /* the node sends the cover to all neighbors */
4.13      SEND $COV(ID_v, cover)$ TO all $nb \in N_v$
4.14      $update := false$
4.15      $cover\_ready := true$
4.16      $S := \emptyset$

Macro $UPDATE\_TBALLS(id, B_td, TBalls)$
U.01  $Rcvd := Rcvd \cup \{id\}$
U.02  if $(id \in Give\_IDs(TBalls))$
U.03  then
U.04      $Remove\_Ball(id, TBalls)$
U.05  endif
U.06  $NewBall(id, B_td, TBalls)$
U.07  $update := true$
```

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:

$$B_v \cap cover \neq \emptyset \Longrightarrow \exists \text{ a pivot in } B_v$$

At the same time, each pivot $p \in cover$, 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 *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 *nbr*, does the following:

- if the message ID is its own ID, discard the message

- otherwise, add the length of the link to *nbr* at the field *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 $p \in cover$)*

A pivot receives another type of messages, the answers of the clients ($ACCEPT\_SUCC$ or $REJECT\_SUCC$.) In case of $ACCEPT\_SUCC$, the sender is added to the data structure H, with distance 0. Also, the *successors* field represents a path in the 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 $N_v$ */
                  Rcvd : set of int
Predicate: select_clients ≡ cover_ready ∧ $ID_v \in$ cover
Macros: ALL_ANSWERS = check whether the pivot has received messages from everybody, in order
                      to end this layer and to go to the next layer, for relabeling
        $RESTART_3$ = restart the layer 4
        UPDATE_H = update H information and returns true if a change was made, otherwise false
                   input: id, dist, nbr: int
Actions:

5.01  (select_clients ∧ reset3 = true)  ⟶
5.02      Rcvd := $ID_v$
5.03      $RESTART_3$

5.04  (select_clients ∧ timeout)  ⟶
5.05      SEND P_DIST($ID_v$, 0) TO all nb ∈ $N_v$
5.06      pivot := $ID_v$
5.07      to_pivot := $ID_v$

5.08  Upon RECEIPT of P_DIST(s, $dist_s$) FROM nbr  ⟶
5.09      if (select_clients ∧ $s \neq ID_v$ ∧ $s \in$ cover)
5.10      then /* update the data structure H */
5.11          $dist_s := dist_s$ + length(link to nbr)
5.12          UPDATE_H(s, $dist_s$, nbr)
5.13          Rcvd := Rcvd ∪ {s}
5.14      endif

5.15  Upon RECEIPT of ACCEPT_SUCC(s, $succ_s$, dest) FROM nbr  ⟶
5.16      if (select_clients ∧ $s \neq ID_v$ ∧ dest = $ID_v$)
5.17      then /* add the sender as a client to H, with the default distance 0 */
5.18          UPDATE_H(id, 0, nbr)
              /* message ACCEPT_SUCC contains a path in the SPT with only its own clients */
5.19          Construct_Tree(SPT, $succ_s$)
              /* check whether all the nodes have replied */
5.20          ALL_ANSWERS
5.21      endif
```

Algorithm 4.2.9 *To_Clients*

```
Upon RECEIPT of $REJECT\_SUCC(s.dest)$ FROM $nbr$ $\longrightarrow$
    if $(select\_clients \wedge s \neq ID_v)$
    then
        if $(dest = ID_v)$
        then
            /* check whether all the nodes have replied */
            $ALL\_ANSWERS$
        else
            SEND $REJECT\_SUCC(s.dest)$ TO $H[dest].neighbor$
        endif
    endif
```

Macro *ALL_ANSWERS*

```
        $Rcvd := Rcvd \cup \{s\}$
        if $(|Rcvd| = n)$
        then /* go to the layer 4 */
            $relabel, reset4 := true$
            $cover\_ready := false$
        endif
```

Macro *RESTART*$_3$

```
$reset4 := true$
$S := \emptyset$
$relabel, reset3 := false$
```

Macro *UPDATE_H*$(id, dist, nbr)$

```
if $(id \notin H)$
then
    $NewCell(H, id, nbr, dist, false)$
    SEND $P\_DIST(id, dist)$ TO all $nb \in N_v \setminus \{nbr\}$
    return true
else
    if $((H[id].neighbor = nbr) \vee (H[id].distance > dist)$
    then /* a new distance to that node */
        $H[id].distance := dist$
        $H[id].neighbor := nbr$
        $H[id].direct := false$
        SEND $P\_DIST(id, dist)$ TO all $nb \in N_v \setminus \{nbr\}$
        return true
    else
        return false
    endif
endif
```

2. *Calculate the nearest pivot (each non-pivot $v \in V - cover$)*

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* $\leftarrow$ that pivot

to_pivot $\leftarrow$ 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 *to_pivot* neighbor) a message *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 *to_pivot* neighbor.

Algorithm 4.2.10 *To_Pivots*

```
Messages: 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

          ACCEPT_SUCC:   sender: the ID of the sender
                         succs: the string of successors' IDs
                         dest : the ID of the destination

Local_variables:  nb, nbr : int /* elements in $N_v$ */
                  min_dist : int /* the minimum distance to the current pivot */
                  update : boolean
                  Rcvd: set of int

Predicate: choose_pivot ≡ cover_ready ∧ $ID_v \notin$ cover
Macro: BETTER_PIVOT  = update some global and local variables in order to choose
                       the nearer pivot of the current node
                       input: min_distance, id, dist, nbr : int

Actions:

6.01  choose_pivot ∧ (reset3 = true ∨ (Rcvd \ cover) ≠ ∅)   ⟶
6.02      Rcvd := ∅
6.03      min_dist := ∞
6.04      $RESTART_3$
```

Algorithm 4.2.11 *To_Pivots*

```
6.05  Upon RECEIPT of $P\_DIST(s, dist_s)$ FROM $nbr$  $\longrightarrow$
6.06      if ($choose\_pivot \wedge s \neq ID_v \wedge s \in cover$)
6.07      then /* update the data structure $H$ */
6.08          $Rcvd := Rcvd \cup \{s\}$
6.09          $dist_s := dist_s + length$(link to $nbr$)
6.10          if ($pivot \notin cover$)
6.11          then
6.12              $pivot := s$
6.13              $to\_pivot := nbr$
6.14              $min\_distance := dist_s$
6.15          endif
6.16          $update := update \vee UPDATE\_H(s, dist_s, nbr)$
              /* check whether we can choose the nearest pivot */
6.17          $BETTER\_PIVOT(min\_distance, s, dist_s, nbr)$
6.18      endif

6.19  $choose\_pivot \wedge Rcvd = cover \wedge update \wedge pivot \in B$  $\longrightarrow$
          /* the closer pivot is selected, so $v$ sends acceptance to it and refusals to the other pivots */
6.20      SEND $ACCEPT\_SUCC(ID_v, ID_v, pivot)$ TO $to\_pivot$
6.21      for all $pp \in cover \setminus \{pivot\}$ do
6.22          SEND $REJECT\_SUCC(ID_v, pp)$ TO $H[pp].neighbor$
6.23      endfor
          /* now activate the next layer */
6.24      $relabel, reset4 := true$
6.25      $update, cover\_ready := false$

6.26  Upon RECEIPT of $REJECT\_SUCC(s, dest)$ FROM $nbr$  $\longrightarrow$
6.27      if ($choose\_pivot \wedge s \neq ID_v \wedge dest \in cover \wedge dest \in H$)
6.28      then /* a message for a pivot */
6.29          SEND $REJECT\_SUCC(s, dest)$ TO $H[dest].neighbor$
6.30      endif

6.31  Upon RECEIPT of $ACCEPT\_SUCC(s, succ_s, dest)$ FROM $nbr$  $\longrightarrow$
6.32      if ($choose\_pivot \wedge dest \in cover \wedge dest = pivot$)
6.33      then /* a message from a client of the same pivot */
6.34          $S := S \cup \{s\}$
6.35          $UPDATE\_H(s, 0, nbr)$
              /* the node adds itself at the end of the message and forward it to the pivot */
6.36          $succ_s := succ_s$+"+" indicates the concatenation operator $ID_v$
6.37          SEND $ACCEPT\_SUCC(s, succ_s, dest)$ TO $to\_pivot$
6.38      endif

Macro $BETTER\_PIVOT(min\_distance, id, dist, nbr)$
B.01  if (($pivot = id \wedge to\_pivot = nbr$) $\vee$ ($min\_distance > dist$))
B.02  then /* update the current pivot information */
B.03      $min\_distance := dist$
B.04      $pivot := id$
B.05      $to\_pivot := nbr$
B.06  endif
```

4.2.5 Pivot_Label and Client_Label

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 *Pivot_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:  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_I : function from E to $2^{L(V)}$ /* 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_clients ≡ relabel ∧ $ID_v$ ∈ cover
Macros: LABEL  = the node v do the relabeling for all the nodes in its SPT tree $S_v$
        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
```

Algorithm 4.2.13 *Pivot_Label*

```
        $RESTART_4$    = 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:

```
7.01  $relabel\_clients \wedge (pivot \notin cover \vee H \neq (B \cup cover \cup S \setminus \{ID_v\}))$  $\longrightarrow$  $RESTART$

7.02  $relabel\_clients \wedge (reset4 = true \vee Rcvd \setminus cover \neq \emptyset))$  $\longrightarrow$
7.03      $REFRESH$
7.04      $reset4 := false$

7.05  $relabel\_clients \wedge$ ($ID_v$ is the first pivot in cover)  $\longrightarrow$
7.06      $REFRESH$
7.07      $LABEL$

7.08  Upon RECEIPT of $CLEAN(s.dest)$ FROM $nbr$  $\longrightarrow$
7.09      if $(dest = ID_v)$
7.10      then /* a new start for some pivot */
7.11          $Rcvd := Rcvd \setminus \{s\}$
              /* node $v$ warns its clients also */
7.12          for all $id \in S \wedge id \neq ID_v$ do
7.13              SEND $CLEAN(ID_v.id)$ TO $H[id].neighbor$
7.14          endfor
7.15      else
7.16          SEND $CLEAN(s.dest)$ TO $H[dest].neighbor$
7.17      endif

7.18  Upon RECEIPT of $RELBL(s.old\_IDs.new\_IDs)$ FROM $nbr$  $\longrightarrow$
7.19      if $(s \in cover \wedge s \neq ID_v \wedge s \notin Rcvd)$
7.20      then
7.21          $V := V \cup \{old\_IDs\}$
7.22          SEND $RELBL(s.old\_IDs.new\_IDs)$ TO all $nb \in N_v \setminus \{nbr\}$
              /* update $H$ corresponding to the new labels*/
7.23          $UPDATE\_IDS(ID_v.s.old\_IDs.new\_IDs)$
              /* store the last value assigned to a node */
7.24          $n := MAX(new\_IDs)$
7.25          if $(n > ni)$
7.26          then
7.27              $ni := n$
7.28          endif
              /* check whether it is our turn */
7.29          if $\neg(\exists pp \in cover$ before $v$ and $pp \notin Rcvd)$
7.30          then
7.31              $LABEL$
7.32          endif
7.33      endif

7.34  $Rcvd = cover \wedge |V| = n$  $\longrightarrow$  $RELABEL\_DONE$
```

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   $(ni, old\_IDs, new\_IDs) := Tree\_Labeling(ID_v, SPT(ID_v), ni)$
L.03   $newID := new\_IDs[ID_v]$
L.04   $UPDATE\_IDS(ID_v, ID_v, old\_IDs, new\_IDs)$
       /* send the new labels to all other nodes */
L.05   SEND $RELBL(ID_v, old\_IDs, new\_IDs)$ TO all $nb \in N_v$
       /* node $v$ is done with its relabeling */
L.06   $relabel := false$

Macro $REFRESH$
R.01   $Rcvd, V, new\_L, new\_I := \emptyset$
R.02   $newlabels := false$
R.03   $ni := 0$
       /* node $v$ warns the other pivots that it will start its relabeling */
R.04   for all $id \in cover \wedge id \neq ID_v$ do
R.05       SEND $CLEAN(ID_v, id)$ 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 $u \in V$ do
D.02       if $(L[u] \neq new\_L[u])$
D.03       then
D.04           $newlabels := true$
D.05       endif
D.06   for all $nbr \in N_v$ do
D.07       if $(I[ID_v, nbr] \neq new\_I[u, nbr])$
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 $u \in V$ do
D.14           $L[u] := new\_L[u]$
D.15       endfor
D.16       for all $nbr \in N_v$ do
D.17           $I[ID_v, nbr] := new\_I[u, nbr]$
D.18       endfor
D.19       $newlabels := false$
D.20   endif
       /* we start a new phase */
D.21   $RESTART_4$
```

Algorithm 4.2.15 *Pivot_Label*

```
Macro RESTART_4
T.01  exist_leader, sent_ball, cover_ready, relabel := false
T.02  reset2, reset3, reset4 := true
T.03  cover, S := ∅
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.08      new_I[current_id, nbr] := new_I[current_id, nbr] ∪ newIDs
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.12      new_I[id, nbr] := new_I[id, nbr] ∪ newIDs[u]
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
                  $new\_L$ : function from $V$ to [1..n] /* new calculated node labeling function */
                  $new\_I$ : function from $E$ to $2^{L(V)}$ /* new calculated arc labeling function */
                  $newlabels$ : boolean /* is $true$ if at least one label is changed */
Predicate: $update\_all \equiv relabel \wedge ID_v \notin cover$
Macro: $UPDATE\_IDS$  = build the interval routing scheme in the current node
                       input: $id, pid$ : int
                              $oldIDs, newIDs$ : array of int
Actions:

$relabel \wedge (pivot \notin cover \vee H \neq (B \cup cover \cup S))$   $\longrightarrow$   $RESTART$

$update\_all \wedge (reset4 = true \vee (Rcvd \setminus cover \neq \emptyset))$   $\longrightarrow$
    $Rcvd, V, new\_L, new\_I := \emptyset$
    $newlabels := false$
    $reset4 := false$

Upon RECEIPT of $CLEAN(s, dest)$ FROM $nbr$   $\longrightarrow$
    if $(s \neq ID_v \wedge dest = ID_v)$
    then /* a message for me */
        $Rcvd := Rcvd \setminus \{s\}$
        /* node $v$ warns its clients also */
        for all $id \in S \wedge id \neq ID_v$ do
            SEND $CLEAN(ID_v, id)$ TO $H[id].neighbor$
        endfor
    endif
    if $(choose\_pivot \wedge s \neq ID_v \wedge dest \in cover \wedge dest \in H)$
    then /* a message for a pivot */
        SEND $CLEAN(s, dest)$ TO $H[dest].neighbor$
    endif

Upon RECEIPT of $RELBL(s, old\_IDs, new\_IDs)$ FROM $nbr$   $\longrightarrow$
    if $(update\_all \wedge s \in (cover \setminus Rcvd))$
    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 $nb \in N_v \setminus \{nbr\}$)
    endif
```

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 $distance = length(link(8,12)) + length(link(12,1)) = 5$. Ordering the nodes in B_1 we have:

B_1

node	12	7	8	13	10
distance	2	3	5	7	8

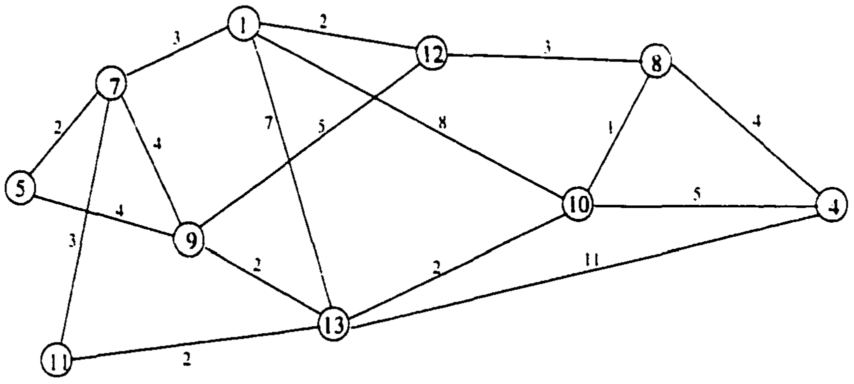

Figure 4.1: A partial t-ball for node 1

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 :

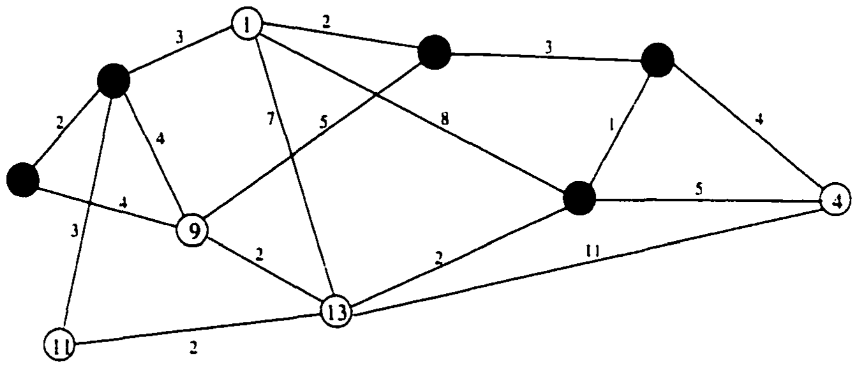

Figure 4.2: The t-ball $B_1(5)$

Next, we prove the correctness of the algorithm.

CHAPTER 5

PROOF OF CORRECTNESS

The construction of the $\mathcal{PIR}$ 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 $k = |cover|$ 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 $\mathcal{SPIR}$ builds in each pivot the SPT as $\mathcal{PIR}$. Based on SPTs, pivots do relabeling, so the IRS is correct and the same as in $\mathcal{PIR}$ 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 *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 v 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 v. Also, each pivot, once it knows all its clients, has to do the relabeling. By doing this in *fair coordinated* manner (Lemma 5.4), we obtain correct labeling functions $\mathcal{R} = (\mathcal{L}, \mathcal{I})$.

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 $cover.S$ are reset to $\emptyset$. Only the guards of layer 1 and 2 are enabled. by setting the appropriate boolean variables to *true* or *false* value. So. it is a restart with *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 *cover* and all the $S_p, p \in cover$. 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 $\mathcal{SPIR}$ constructs a $\mathcal{PIR}$ 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 *Calculate_Ball* and some guards in *Error_Correction*.

Adding nodes to B is done automatically. and the macro $UPDATE$ in *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 $UPDATE$ in *Calculate_Ball*:

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

Consider m as the initial value of $H_v[u].distance$. 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 $H_v[u].distance$ converges to the shortest distance (condition $H[id].distance > dist$ in line U.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 $H_v[u].distance$ converges to the shortest distance.

The value m is stored as the distance from u to v through a neighbor $H_v[u].neighbor$. 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:

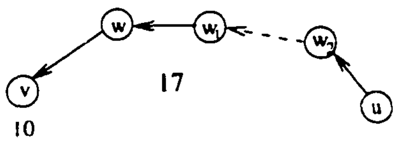

Figure 5.1: A correct distance replaces the old one

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: $H_v[u].distance = 10$
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 : $H_v[u].distance = 17$
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 U.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 U.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 ($u \notin N_v$), 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 ($w \in N_v$):*

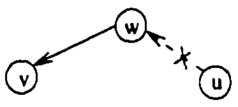

Figure 5.2: The node u reached v through w but the path $u \rightarrow w$ is disconnected

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 $u \in B_v \Rightarrow$ 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 ($H_v[u].direct = true$):

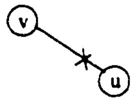

Figure 5.3: A crash in the direct neighborhood

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

We have a situation like this:

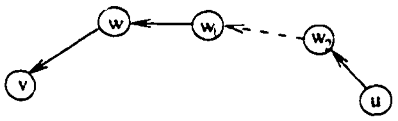

Figure 5.4: The down node u reached v through w

$\exists w_k$ such that $u \in N_{w_k} \wedge H_{w_k}[u].direct = true \Rightarrow B_{w_k}$ and H_{w_k} get updated. Recursively, B_w and H_w get updated. □

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:

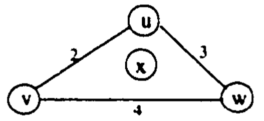

Figure 5.5: A cycle in delivering of dimension 3

and we have the following values :

$H_v[x].neighbor = w$: 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 $x \in B_u \wedge H_u[x].neighbor = v \Rightarrow x \in B_v$ 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 $H_v[x].distance = length(v, w) + H_w[x].distance$ and it removes the eventual cycles.

To understand how it works, consider the following example:

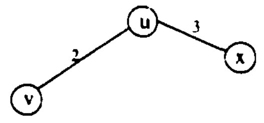

Figure 5.6: A correct situation

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 $H_v[x].distance = 12$: $\Rightarrow H_w[x].distance$ must be $12 - 4 = 8 \Rightarrow H_u[x].distance$ must be $8 - 3 = 5 \Rightarrow H_v[x].distance$ 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 $\mathcal{SPIR}$ 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 analyse 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 $RESTART_4$ gets executed, we make sure to keep in H whatever is in B. So, the condition $(B \setminus H \neq \emptyset)$ 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 $(B = \emptyset \wedge (cover \cup S) \neq \emptyset)$ 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 $(cover = \emptyset \wedge S \neq \emptyset)$ 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 *cover*. Also. whenever an ID is added to S. it is added to H also. Therefore the condition $H \not\subseteq (B \cup cover \cup S)$ is *false*.

When macro *RESTART* is executed. the variable *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 *to_pivot* crashes (or the link to that neighbor crashes.) In this case. in the macro *REMOVE* another neighbor is chosen (line R.05). so *to_pivot* $\in (N_v \cup \{ID_v\})$ remains *true*.

(iii) the node v is selected as a pivot. so *to_pivot* has (and keeps) the value ID_v. so the condition *to_pivot* $\in (N_v \cup \{ID_v\})$ remains *true*.

(iv) the node v is not selected as a pivot. so it has to choose its nearest pivot. The variable *min_distance* keeps the lowest distance received toward any of the pivots. so whenever a lower distance toward a pivot is received. *to_pivot* is changed to that neighbor which sent this information. therefore *to_pivot* $\in (N_v \cup \{ID_v\})$ remains *true*. □

Property 5.1.10 *Once the macro RESTART is executed. the guard 1.04 of the algorithm Error_Correction (keeping correct data about the neighbors) remains false.*

Proof. The field *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 *direct* of that element is set to *false* (lines U.12. U.18. U.25. of the macro *UPDATE* in the algorithm *Calculate_Ball*. lines U.03. U.11 of the macro *Update_H* in the algorithm *To_Clients*.)

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

The field *distance* can change when a shorter distance is detected. But at that time.

the field *direct* is set to *false* automatically (lines U.12 of the macro *UPDATE* in the algorithm *Calculate_Ball*, line U.11 of the macro *Update_H* in the algorithm *To_Clients*.)

Taking an example:

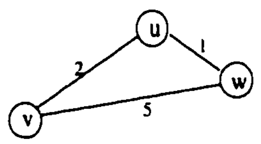

Figure 5.7: A shorter distance through another neighbor

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

$H[w].neighbor = w$ $\quad$ $H[w].distance = 5$ $\quad$ $H[w].direct = true$

If node v detects a shorter distance to node w through node u, the values are changed in the macros *UPDATE* (lines U.10-12) or *Update_H* (lines U.09-11) to:

$H[w].neighbor = u$ $\quad$ $H[w].distance = 3$ $\quad$ $H[w].direct = false$ □

Property 5.1.11 *The guard 2.02 in the algorithm Calculate_Ball (keeping data about unknown nodes, whose DIST messages have not been yet received) remains false.*

Proof. The set *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 *RESTART* when such a node is detected. After *RESTART* gets executed and B becomes $\emptyset$, $Rcvd\_IDs := \{ID_v\}$. From here on, *Rcvd_IDs* will start storing only the IDs of the nodes which have sent messages and maybe have changed the set B. So, $B \subseteq Rcvd\_ID$, so the guard 2.02 remains *false*. □

After *RESTART* is executed, the variable *pivot* is set to the node ID. If the node is part of the *cover*, we make sure that *pivot* and *to_pivot* is the node ID. But if the node is not part of the cover, *pivot* is set arbitrarily to one of the pivots (lines 6.10-12 in the algorithm *To_Pivots*.) Starting from this point, *pivot* is modified to the closest pivot whose message has been received (macro $BETTER\_PIVOT$.)

Property 5.1.12 *If v is a pivot, in the algorithm 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 *pivot* is set to its own ID (line 5.06 in the algorithm *To_Clients*), so $pivot \in cover$. The algorithm *Pivot_Label* runs only for the pivots. So, when *relabel* becomes *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 $H = (B \cup cover \cup S)$. □

Property 5.1.13 *If v is not a pivot, in the algorithm 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 *Client_Label* runs only for the non-pivot nodes (a client of some pivot.) Thus, when *relabel* becomes *true*, this means that v has received messages from all the pivots and successors, and H contains this information. So, *pivot* is set to the nearer pivot (macro $BETTER\_PIVOT$ in the algorithm *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 *To_Pivots*.) So, the condition $relabel \wedge (pivot \notin cover \vee H \neq (B \cup cover \cup S))$ is *false*. □

We have shown that all the possible guards which can determine a re-execution of $RESTART$ remains *false* after a $RESTART$ has been executed, so we have at most one $RESTART$ in every execution. □

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 $\mathcal{PIR}$ 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 *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 *ball_sent* is set to *true* (line 2.25.)

A node. even if it is not done calculating its t-ball. forwards eventually t-balls received toward the *leader* (line 3.03.) The leader collects the t-balls and once its t-ball is calculated (*ball_sent* = *true*) and all other t-balls are received. it calculates a cover based on these t-balls. using a *greedy* method. Whenever a new t-ball is received. *leader* checks whether that node did not send one before. If the node has sent an old t-ball. the *leader* replaces it with the new one (macro *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 *cover_ready* becomes *true* (lines 3.12. 4.15) for the third level.

In the third level. the nodes are divided into ***pivots*** and *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 *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*;

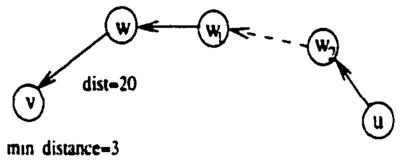

Figure 5.8: Increasing min_distance

Now, it is the turn of the node v to replace the old value of *min_distance* which is 3 with the newer one, 17. □

5.3 Relabeling

We already know that a non-pivot node v always chooses its pivot as a node from the *cover*. But that pivot should be also in its t-ball: $cover \cap B_v \neq \emptyset \Rightarrow \exists p \in cover \wedge p \in B_v$. 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 $Rcvd = cover$ is necessary to be *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 *cover_ready* becomes *false* (lines A.05 in the macro *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 *relabel* and *reset4* to *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 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 *Pivot_Label* imposes the restriction of at most one message from each pivot. We can have no more messages accepted from a pivot p if $p \in Rcvd$ 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* $:= \emptyset$ 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 $CLEAN$ 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 $CLEAN$ message to determine the node v to remove p from *Rcvd*, so the new relabels of p will be accepted by v.

□

A node accepts only one message $RELBL$ is order to prevent the network to be flooded. For a pivot, we need to prove that the relabeling process respects the $\mathcal{PIR}$ preprocessing step of [EGP98]. Or simply, the fair-coordination among the pivots dictates when each pivot does its relabeling.

Lemma 5.4 (Fair-coordination) *The pivots relabeling is done in fair-coordinated manner.*

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

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 $CLEAN$ first. Because we have FIFO channels when the message $RELBL$ containing the relabel reaches a node, we know that $CLEAN$ was previously received and the node is able to accept the relabels. $CLEAN$ 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 *relabel* $=$ *true*. After receiving $CLEAN$ 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 *CLEAN* messages are restricted to be sent be its pivot. *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. □

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 *cover*, S are reset to $\emptyset$. Only the guards of layer 1 and 2 are enabled, by setting the appropriate boolean variables to *true* or *false* value. Thus it is a restart with *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 *cover* and all the $S_p, p \in cover$, 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 $p \in cover$, calculated in the algorithm *To_Clients*.

Because the property of ***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 $\mathcal{L}_I$. $\mathcal{L}_I = I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5$ 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 : $cover = Calculate\_cover\{B_v(t) | v \in V\}$ (the cover calculated by the algorithm

Calculate_Cover is the cover for all t-balls.)

I_3 : $pivot = p_v. \forall v \in V - cover$ (the nearer pivot calculated in the algorithm *To_Pivots* is the nearest pivot in terms of the distance $\prec_v$. for each non-pivot node.)

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

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

I_5 : $(L. I)$ constructed in the macro *UPDATE_IDS* is the $\mathcal{PIR}$ 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. □

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 *I_5 is a closed attractor for C.*

Proof. Consider the moment when *relabel = true* for every node and *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 $\mathcal{PIR}$. and by accepting one message per pivot (Lemma 5.3.) So. in $O(kd)$. where $k = |cover|$. 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]. $k = |cover| \leq \sqrt{n(1 + \log n)}$. so the stabilization time becomes $O(d\sqrt{n(1 + \log n)})$ time units. □

Theorem 5.4.1 (Closure and convergence of $\mathcal{L_I}$) *The distributed algorithm constructs a $\mathcal{PIR}$ scheme in polynomial time. The characteristics of the $\mathcal{PIR}$ 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 $\mathcal{SPIR}$. 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 to find efficient self-stabilizing solutions (more efficient than the existing ones) to the above problems.

BIBLIOGRAPHY

[ABNL$^+$89] B. Awerbuch. A. Bar-Noy. N. Linial. . and D. Peleg. Compact distributed data structures for adaptive routing. In *STOC89 Proceedings of the 21th Annual ACM Symposium on Theory of Computing*. volume 2. pages 230-240. 1989.

[ABNLP90] B. Awerbuch. A. Bar-Noy. N. Linial. and D. Peleg. Improved routing strategies with succint tables. *Journal of Algorithms*. 11:307-341. 1990.

[AG93] A. Arora and M. G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*. 19:1015-1027. 1993.

[AKM$^+$93] B. Awerbuch. S. Kutten. Y. Mansour. B. Patt-Shamir. and G. Varghese. Time optimal self-stabilizing synchronization. In *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. pages 652-661. 1993.

[AP90] B. Awerbuch and D. Peleg. Sparse partition. In *FOCS90 Proceedings of the 30st Annual IEEE Symposium on Foundations of Computer Science*. pages 503-513. 1990.

[APSV91] B. Awerbuch. B. Patt-Shamir. and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*. pages 268-277. 1991.

[APSV94] B. Awerbuch. B. Patt-Shamir. and G. Varghese. Bounding the unbounded. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*. 1994.

[DH97] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoritical Computer Science*. 3(4). 1997.

[Dij74] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643-644. 1974.

[Dij82] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Selected Writings of Computing: A Personal Perspective*, pages 41-46. 1982.

[Dol97] S. Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2):122-127. 1997.

[Dol00] Shlomi Dolev. *Self-Stabilization*. The MIT Press. 2000.

[EGP98] T. Eilam. C. Gavoille. and D. Peleg. Compact routing schemes with low stretch factor. Technical report. LaBRI. Universite Bordeaux. Weizmann Institute of Science. 1998.

[Gou98] M. G. Gouda. *Elements of network protocol design*. John Wiley & Sons. Inc.. 1998.

[GP99] C. Gavoille and D. Peleg. The compactness of interval routing. *SIAM Journal on Discrete Mathematics*. 12:459–473. 1999.

[GS95] M. G. Gouda and M. Schneider. Maximum flow routing. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*. pages 2.1–2.13. 1995.

[KP93] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*. 7:17–26. 1993.

[LAJ99] C. Labovitz. A. Ahuja. and F. Jahanian. Experimental study of internet stability and wide-area network failures. In *Proceedings of FTCS99*. 1999.

[LT87] J. Leeuwen and R. B. Tan. Interval routing. *Computer Journal*. pages 298–307. 1987.

[Mas95] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*. pages 1.1–1.15. 1995.

[PU89] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. In *Journal of the ACM*. volume 36. pages 510–530. 1989.

[SK85] N. Santoro and J. Khatib. Labeling and implicit routing in the networks. In *Computer Journal*. volume 28. pages 5–8. 1985.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press. 1994.

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

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

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

Thesis Examination Comittee:
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.