Self-stabilizing interval routing algorithm with low stretch factor

Sripriya Sundaram
University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1263

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

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

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

UMI®
SELF-STABILIZING INTERVAL ROUTING ALGORITHM
WITH LOW STRETCH FACTOR

by

Sripriya Sundaram

Bachelor of Engineering
University of Madras, India
1996

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
May 2001
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

July 17th, 2000

The Thesis prepared by

SRIPRIYA SUNDARAM

Entitled

SELF-STABILIZING INTERVAL ROUTING ALGORITHM WITH LOW STRETCH FACTOR

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

MASTERS OF SCIENCE

Ajoy Kumar Dutta
Examination Committee Chair

Dean of the Graduate College

Keeley Koegel
Examination Committee Member

John T. Minor
Examination Committee Member

M.B. Tikhomirov
Graduate College Faculty Representative
ABSTRACT

Self-stablizing Interval Routing Algorithm with Low Stretch factor

by

Sripriya Sundaram

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

A compact routing scheme is a routing strategy which suggests routing tables that are space efficient compared to traditional all-pairs shortest path routing algorithms. An Interval Routing algorithm is a compact routing algorithm which uses a routing table at every node in which a set of destination addresses that use the same output port are grouped into intervals of consecutive addresses. Self-stabilization is a property by which a system is guaranteed to reach a legitimate state in a finite number of steps starting from any arbitrary state. A self-stabilizing Pivot Interval Routing (PIR) algorithm is proposed in this work. The PIR strategy allows routing along paths whose stretch factor is at most five, and whose average stretch factor is at most three with routing tables of size $O(n^{3/2} \log_2^{3/2} n)$ bits in total, where $n$ is the number of nodes in the network. Stretch factor is the maximum ratio taken over all source-destination pairs between the length of the
paths computed by the routing algorithm and the distance between the source and the
destination. PIR is also an Interval Routing Scheme (IRS) using at most \( 2n(1+lnn)^{1/2} \)
intervals per link for the weighted graphs and \( 3n(1+lnn)^{1/2} \) intervals per link for the
unweighted graphs. The preprocessing stage of the PIR algorithm consists of node­
labeling and arc-labeling functions. The node-labeling function re-labels the nodes with
unique integers so as to facilitate fewer number of intervals per arc. The arc-labeling is
done in such a fashion that the message delivery protocol takes an optimal path if both
the source and the destination are located within a particular range from each other and
takes a near-optimal path if they are farther from each other.
## TABLE OF CONTENTS

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

ACKNOWLEDGEMENTS ........................................................................................................ vii

CHAPTER I INTRODUCTION ............................................................................................... 1
  Distributed System ................................................................. 1
  Fault Tolerance ........................................................................ 2
  The Routing Problem ........................................................... 5
  Related work ........................................................................ 7
  Contributions ........................................................................ 8

CHAPTER II PRELIMINARIES .......................................................................................... 10
  Self-stabilizing System .......................................................... 10
    System ................................................................. 10
    Programs ......................................................... 11
    Predicates .................................................... 12
    Self-Stabilization .................................................... 12
  Pivot Interval Routing Scheme ........................................... 13
  Specification of the Pivot Interval Routing Scheme .......... 15

CHAPTER III SELF-STABILIZING INTERVAL ROUTING .......................................... 16
  Outline of Algorithm SPIR .................................................. 16
  Data Structures for Algorithm SPIR .................................... 18
  Algorithm SPIR in detail ..................................................... 19
    Algorithm Min_BFS_trees.............................................. 19
    Algorithm CalcSize_TreeClts ....................................... 19
    Algorithm Conv_BcstSzs ............................................. 23
    Algorithm Number_Nodes ............................................ 27
    Algorithm Label_Arcs ............................................... 30
  The Routing Module ......................................................... 38

CHAPTER IV PROOFS OF CORRECTNESS ................................................................... 39
  Correctness of the Node Numbering Phase ....................... 39
  Correctness of the Arc Labeling Phase .............................. 40
  Complexity Analysis ......................................................... 41

CHAPTER V CONCLUSION .............................................................................................. 44
ACKNOWLEDGEMENTS

To my advisor Dr. Ajoy Kumar Datta, for his continual support and encouragement throughout the entire length of my study. My sincere thanks to him for always being there for me as a friend, philosopher and guide.

To Dr. John Minor, Dr. Kazem Taghva and Dr. Mohamed B Trabia for serving on my thesis committee.

To all the faculty in the department of Computer Science. Everybody has an important role in stepping up my knowledge and confidence in different areas of Computer Science.

To my family. My dad is the reason for everything I am in my career and my mom is the reason for everything I am as a person. My little brother has been there for me with all his support in every decision I make.

To my husband, Natarajan Krishnan. He is the best thing that has ever happened to me. I don’t feel lonely anymore.

To all my friends in the UNLV and in the rest of the world for all the great times we have had together. I will cherish them in my heart for the rest of my life.
CHAPTER 1

INTRODUCTION

1.1 Distributed System

A distributed system is an interconnected collection of autonomous processes. Distributed systems have evolved using message passing as their main method of communication. Another method of communication that has been gaining popularity in today's world is the shared memory mechanism. Today's multiprocessor systems use this mechanism to a greater extent. Some of the reasons for the popularity of the distributed shared memory systems are that large or complex data structures may easily be communicated, and shared memory also gives transparent process-to-process communication.

The use of distributed systems has many advantages over the traditional sequential systems. By virtue of resource sharing and openness properties, the distributed systems promote increased resource sharing and facilitate a more modular design of the network. The other advantages include increased performance through better usage of system resources and increased reliability through replication. A sophisticated approach to the design of distributed algorithms includes considering the chance that the nodes and/or links may fail. This gives the distributed systems, the quality of being fault tolerant and
this is the topic of interest in our next section. The price we pay to enjoy all these advantages is the extreme complexity to build a distributed system because of its lack of global knowledge and possible aynchrony.

1.2 Fault Tolerance

One of the key characteristics of the distributed systems is fault tolerance. The traditional way of achieving fault tolerance is to use redundant hardware resources. In this way, if a hardware resource fails, the redundant hardware is switched to help the system to continue with the activities. So in case of link failures, fault tolerance is achieved by some method of interconnecting the networks such that if a link fails, redundant links come into play helping not to lose the information floating in the network.

Another way of achieving fault tolerance has gained a lot of popularity because it is one of the most inclusive and unified approaches to fault tolerance for distributed systems. It is the paradigm of *Self-Stabilization*. Informally, a protocol is said to be self-stabilizing if its specification does not require a certain “initial configuration” to be imposed on the system to ensure correct behavior of the protocol. Alternatively, self-stabilization is the property that guarantees that a system placed in an arbitrary state will return to a legitimate state within a finite number of state transitions. Contrary to non-stabilizing systems, self-stabilizing systems are capable of recovering automatically from illegitimate global states without manual intervention. The user doesn’t even know that a transient failure has occurred when it does. These properties make the self-stabilizing
systems more reliable and more powerful than the non-stabilizing systems. This also saves the cost of using redundant hardware to achieve fault-tolerance.

From a theoretical point of view, a self-stabilizing protocol is “cleaner”, since the disturbing need to specify an initial state is eliminated. Secondly, allowing a protocol to be initialized in an arbitrary state can be viewed as an abstraction of a strong fault tolerant property. In particular, a self-stabilizing protocol can recover from a memory corruption, a property that none of the benign failure models can handle. The self-stabilization model is especially appropriate for the case of “infrequent catastrophes”: every once in a (long) while the system may crash, resulting in an arbitrary state. A self-stabilizing protocol guarantees that eventually the system will regain stability in some legitimate state. Hence, relevant complexity parameters of a self-stabilizing solution are stabilization time, and overhead in time/communication when the system is operating in a “regular mode” (that is, when all states are legitimate), as compared to a “conventional” protocol solving the same problem.

Dijkstra's notion of self-stabilization [2], which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault tolerance. Following Dijkstra’s introduction to the notion of self-stabilization, very few papers were published in this area in ten years or so. In an invited address, Lamport [41, 43] mentioned that this is the most brilliant work of Dijkstra and regarded self-stabilization to be a very important concept in fault tolerance and to be a very fertile field for research.

Several approaches of achieving self-stabilization have been proposed since it began to attract the attention of the researchers in the late 1980s [37, 38, 39, 40]. One very good source for a perspective of the research in self-stabilization until 1992 was the survey on
Self-Stabilization by Marco Schneider [10]. Many approaches to achieving self-stabilization exist [1, 3, 4, 5, 9, 11, 12]. The main aspects of focus in the area of self-stabilization have been time optimality, memory efficiency and other limitations. Many non-stabilizing algorithms have been made self-stabilizing such as in the area of mutual exclusion [2, 15], token-passing protocols [6, 13, 14], spanning tree algorithms [7, 8, 16] and several other graph algorithms. While most of the work was directed at self-stabilization of specific tasks, some work was devoted to designing general algorithmic transformers that take a protocol as input, and produce as their output a self-stabilizing version of that protocol. These transformers typically exhibit trade-offs between their generality and the efficiency of the resulting protocols. One such general transformation is given by Katz and Perry [50], where they show how to compile an arbitrary asynchronous protocol into a self-stabilizing equivalent. But this method of self-stabilization was highly inefficient in terms of time, space and communication. AwerBuch et al [2] suggested a transformer that has improved efficiency in terms of time and space with only a small increase in communication cost and this transformer can be applied to many practical problems, including spanning tree construction. Another popular method of self-stabilization is to reset the system to a predefined global state, when deemed necessary [11, 38]. Ideally, resetting a distributed system to a global state implies resuming the execution of the system starting from a given state. With this characterization, however, each reset of a distributed can be achieved only by a “global freeze” of the system. This seems rather limiting and, in many applications, more strict than needed. Therefore, [38] adopts a characterization where resetting a distributed
system to a given global state implies resuming the execution of the system from a global state that is reachable, by some system computation, from the given global state.

Self-stabilization is, indeed, a fertile area of research. We have tried our best to mention all the important work in the area of self-stabilization in this section.

1.3 The Routing Problem

Routing in distributed systems can be described as locally constructing, at each processor \( i \), a uniform function \( \Gamma(i) \) such that given an input destination and a received message, the output image of \( \Gamma(i) \) is a suitable neighbor of processor \( i \) to forward the message to.

The classical routing strategy [46] is based on schemes that keep in each node, a full routing table, which specifies an output port for every destination. The use of these routing tables uses shortest paths but requires \( O(n \log n) \) bits at each node.

In a parallel and distributed system, as more processors are added to increase the computing power, the underlying communication network needs to scale favorably along with the expansion. The routing methods used should also be simple and dynamically adjustable with expansion. That's why more and more emphasis is given to universal routing [42, 44] methods, which do not rely on underlying topology and allow the underlying network to be quite arbitrary. Though the classical shortest path routing strategy is a universal scheme, its memory requirements makes it almost impractical when dealing with large networks. This gives rise to the need for simple compact routing protocols that are scalable with the growth of networks.
The compact routing scheme that is very popular is the interval routing [18, 20, 22, 33, 47] scheme which was introduced in [26, 45]. This is a routing strategy where for every output port, the destination nodes to be reached through that port are all grouped together. This makes this routing strategy very space-efficient. However, very little is known about the topological properties that must satisfy a network to support an interval routing function with particular constraints. Additionally, these often use a path length greater than optimal. The efficiency of the compact routing scheme is measured in terms of stretch factor introduced by Awerbuch et al [2]. Stretch factor is the maximum ratio between the length of the path traversed by a message using the compact routing scheme and that of the shortest path between its source and destination.

There has been quite a lot of work done in the area of interval routing. Main aspects of interval routing under consideration are number of intervals used per arc (one or many), space-efficiency, other lower bounds in terms of stretch factor and topological constraints [18, 19, 22, 34]. A nice survey on interval routing schemes can be found in [27]. Our work is on an interval routing strategy called Pivot Interval Routing [24]. Pivot interval routing (PIR) strategy is in between shortest path routing and the other compact routing algorithms in the sense it has some features of both. This algorithm is intended to reduce the complexity inherent in some compact routing strategies that are similar in nature and also to reduce the space complexity to a major extent. We will go into the details of this algorithm in the sections to come.
1.4 Related work

Though a lot of research has been done in the area of compact routing, the area of compact routing that is more related to our work is hierarchical routing scheme [17, 34, 35, 36, 49].

For routing in large networks, the reduction of routing information is realized through a hierarchical partitioning of the network nodes. Basically, an $m$-level partitioning of a set of nodes, where $m \geq 2$, consists of grouping the nodes into 1$^{st}$ level clusters, which in turn are grouped into 2$^{nd}$ level clusters, etc. This operation continues in a bottom up fashion, finally grouping the $m - 2^{nd}$ level clusters into $m - 1^{st}$ level clusters whose union constitutes the $m^{th}$ level cluster. A message from a source is sent to the next upper level cluster if the destination is not in that cluster and so on until the cluster that has the destination is reached and then the message is delivered to the destination.

The hierarchical routing scheme in [35] provides an optimal efficiency-space tradeoff. It shows that for every graph and for every integer $k \geq 1$, it is possible to construct a hierarchical routing scheme with a stretch factor of $O(k)$ which uses a total of $\Omega(k^3n^{1+1/k}\log n)$ bits. But the disadvantages are it is not name independent and also this scheme does not bound the local memory requirements of a node. There are other hierarchical routing methods [17, 36] that avoid these problems at the price of efficiency-space optimality. [17] achieves $O(kn^{1/k}\log n)$ bits of memory per node with a stretch factor of $O(k^29^k)$ while [36] achieves stretch factor of $O(k^2)$ and uses $O(kn^{1/k}\log n\log D)$ bits per node, where $D$ is the weighted diameter of the network.
The major disadvantage of all the proposed hierarchical routing strategies is that they involve complex decision function at the nodes. The delay in sending the messages to the destination is mainly because of the time taken to make the decision as to where the message is to be sent next, and not the propagation delay. This becomes a major bottleneck in the case of high-speed networks where the implementation of hierarchical routing is impractical. Therefore, simple routing schemes like *Pivot Interval Routing*, which could be implemented in high-speed networks, may be preferable. This scheme uses a direct routing scheme where the message header contains only the destination information, and the decision making at the node is very simple.

1.5 Contributions

In this paper, we propose a self-stabilizing pivot interval routing strategy *SPIR*. In this protocol, the network is divided into balls, called t-balls, where t is the number of nodes in the ball. This algorithm is constructed in such a way that a message sent from a node travels an optimal path if the destination is in the same ball as the node. Otherwise, travels the shortest path to the destination ball to a node called the pivot, whose significance will be described later, and from there it travels the shortest path to the destination.

*PIR* is similar to the hierarchical routing scheme in the sense it involves m-level message passing, where m = 2, in case the source-destination pair are being in different balls. Also, *PIR* has the same stretch factor and memory requirements as the hierarchical routing scheme. But the hierarchical routing scheme involves major decision making in the nodes which makes its implementation almost impossible in high-speed networks.
Whereas, \textit{PIR} uses the direct routing scheme by virtue of which very simple decision making is done at a node. It employs the ‘shoot and forget’ method, where only the destination name is required to find the suitable port to which the message should be directed.

In \textit{SPIR}, we use an underlying depth-first circulation algorithm and also the shortest path spanning tree algorithm. The algorithm is divided into two phases namely the node-naming phase and the arc-labeling phase. Node-naming phase is done in such a way that the node names are unique, which helps in reducing the number of intervals required in the arc-labeling phase. Then, the arc-labeling is done in such a way that the message traverses a path as suggested by the \textit{PIR} algorithm. These phases use the depth-first token circulation and shortest-path spanning tree algorithms.

\textit{SPIR} algorithm has a space complexity of $6mn^{1/2}(1+\log n)^{1/2}\log n$ at each node and $O(n^{3/2}\log n^{3/2})$ bits in total for storing interval information and the average stretch factor is at most three. The time complexity depends on the underlying depth-first token circulation and shortest path spanning tree algorithms. After these algorithms stabilizes, \textit{SPIR} takes $O(n)$ steps to stabilize.

The remainder of the documentation is organized as follows: Chapter 2 discusses the preliminary model of the system used in this work, along with some important definitions. Chapter 3 presents the proposed \textit{SPIR} algorithm. Chapter 4 presents the proofs of correctness for the \textit{SPIR} algorithm. Finally, conclusions and some future research directions are discussed in Chapter 5.
CHAPTER 2

PRELIMINARIES

In this chapter, we define the distributed systems and programs considered in this work, and state what it means for a protocol to be self-stabilizing. We then introduce some definitions related to the pivot interval routing algorithm.

2.1 Self-stabilizing System

2.1.1 System

A distributed system is a uni-directed connected graph, $S = (V, E)$, where $V$ is a set of processors ($|V| = n$, $n$ is the number of processors in the network) and $E$ is the set of bi-directional communication links. We consider networks which are asynchronous and rooted, i.e., all processors, except the root are anonymous. We denote the root processor by $r$. The numbers $1, .., n$, are used to identify the processors to present our ideas here, but no processor, except the root (identified by $r$), has any identity. A communication link $(p,q)$ exists iff $p$ and $q$ are neighbors. Each processor $p$ maintains its set of neighbors, denoted as $N_p$. We assume that $N_p$ is a constant and is maintained by an underlying protocol.
2.1.2 Programs

Each processor executes the same program except the root \( r \). The program consists of a set of \textit{shared variables} (henceforth referred to as variables) and a finite set of actions. A processor can only write its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of \( p \) can be accessed by \( p \) and its neighbors.

Each action is uniquely identified by a label and is of the following form:

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

The guard of an action in the program of \( p \) is a boolean expression involving the variables of \( p \) and its neighbors. The statement of an action of \( p \) updates zero or more variables of \( p \). An 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 statement of an action, if executed are done in one atomic step. The atomic execution of an action of \( p \) is called a step of \( p \).

The state of a processor is defined by the values of the variables. The state of a system is a product of the states of all processors (\( \in V \)). In the sequel, we refer to the state of a processor and system as a \textit{(local) state and configuration}, respectively. Let a distributed protocol \( P \) be a collection of binary transition relations denoted by \( \rightarrow \), on \( C \), the set of all possible configurations of the system. A \textit{computation} of a protocol \( P \) is a maximal sequence of configurations \( e = (\xi_0, \xi_1, \ldots, \xi_i, \xi_{i+1}, \ldots) \), such that for \( i \geq 0 \), \( \xi_i \rightarrow \xi_{i+1} \) (a \textit{single computation step}) if \( \xi_{i+1} \) exists, or \( \xi_i \) is a terminal configuration. \textit{Maximality} means that the sequence is either infinite, or it is finite and no action of \( P \) is enabled in the final
configuration. All computations considered are assumed to be maximal. During a computation step, one or more processors execute a step, and a processor may take at most one step. This execution model is known as the distributed demon. We use the notation \( \text{Enable}(A, p, \gamma) \) to indicate that the guard of the action \( A \) is true at processor \( p \) in the configuration \( \gamma \). A processor \( p \) is said to be enabled at \( \gamma (\gamma \in C) \) if there exists an action \( A \) such that \( \text{Enable}(A, p, \gamma) \). We assume that our demon is a weakly fair daemon, meaning that if a processor \( p \) is continuously enabled, then \( p \) will be eventually chosen by the daemon to execute an action. The set of computations of a protocol \( P \) in a system starting with a particular configuration \( \alpha \in C \) is denoted by \( E_\alpha \). The set of possible computations of \( P \) in system \( S \) is denoted as \( e \). A configuration \( \beta \) is reachable from \( \alpha \), denoted as \( \alpha \rightarrow \beta \), if there exists a computation \( e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots) \in E_\alpha (\alpha = \gamma_0) \) such that \( \beta = \gamma_i (i \geq 0) \).

2.1.3 Predicates

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

2.1.4 Self-Stabilization

We use the following term, \( \text{attractor} \) in the definition of self-stabilization.
Definition 2.1 (Attractor). Let $X$ and $Y$ be two predicates of a protocol $P$ defined on $C$ of system $S$. $Y$ is an attractor for $X$ if and only if the following condition is true:

$$\forall \alpha \models X : \forall e \in E_\alpha : e = (\gamma_0, \gamma_1, \ldots) \quad \exists i \geq 0, \forall j \geq i, \gamma_j \models Y$$

We denote this relation as $X \triangleright Y$.

Definition 2.2 (Self-stabilization). The protocol $P$ is self-stabilizing for the specification $SP_P$ on $C$ if and only if there exists a predicate $L_P$ (called the legitimacy predicate) defined on $C$ such that the following conditions hold:

1. $\forall \alpha \models L_P : \forall e \in E_\alpha : \models \forall e \models SP_P$ (correctness).
2. $true \triangleright L_P$ (closure and convergence).

2.2 Pivot Interval Routing Scheme

Before we get into the details of how exactly this algorithm works, some basic definitions that will be referred to in the subsequent sections, are introduced below:

Definition 2.3 (t-ball). For every node $v$, we can order all the nodes of the graph w.r.t. $v$ by increasing distance from $v$, breaking ties by increasing node identifications. Formally, $x \Rightarrow y$ if and only if $dist_G(x, v) < dist_G(y, v)$ or $dist_G(x, v) = dist_G(y, v)$ and $x < y$. The $t$-ball $B_v(t)$ of $v$, is the set of the set of the first $t$ nodes according to the node ordering w.r.t. $v$.

Since each node has a $t$-ball of its own, the whole network has $n$ $t$-balls, where $n$ is the number of nodes in the network. The significance of $t$-balls is to allow the routing strategy to adopt a 2-layered message passing.
Definition 2.4 (Cover). Consider a collection $H$ of subsets of size $t$ elements from a set $J$. A set $P \subseteq J$ is said to cover the collection $H$ if for every $A \in H$, $A \cap P \neq \emptyset$. In the PIR algorithm, we call $P$, a set of pivots.

In the PIR algorithm, the cover is formed by including at least one node from each ball. This doesn’t imply that there must be greater than or equal to $n$ number of nodes in the cover because a node can belong to several $t$-balls.

Definition 2.5 (Client). Consider that there is a cover $P$ for the $t$-balls in a graph $G$. In the PIR algorithm, a node from the $P$, say $j$ that is nearest to node $i$ is assigned to $i$ as $i$’s pivot. Now $i$ is said to be the client of $j$.

Definition 2.6 (Interval). Every arc label from a node $i$ to the neighbor $j$ is denoted by $I(i, j)$. $I$ is called an interval for the edge $(i, j)$.

Now that we have seen the basic definitions involved in the PIR strategy, let us look at how exactly this algorithm is intended to work. First, $t$-balls for all the nodes are collected. Then, a cover is formed from the collection of $t$-balls. Let $P$ be a cover for the collection of $t$-balls of the nodes, $\{B_v(t) \mid v \in V\}$, where $V$ is a set of all the nodes in the network. To every node $v$, a pivot, $p(v)$ is assigned, where $p(v)$ is the nearest node to $v$ in $P$. For every pivot $p \in P$, let $S_p = \{ v \mid p(v) = p \}$ be its set of clients. Now finally, the node-labeling and arc-labeling are done in such a fashion that a message with source $u$ and destination $v$ will be routed as follows. If the destination $v$ is in $u$’s $t$-ball, then the message will traverse a shortest path from $u$ to $v$. Otherwise, it will traverse (in worst case) a shortest path to the pivot nearest to $v$, and then a shortest path from that pivot to $v$ itself. We will see how node-labeling and arc-labeling are done in sections to come.
2.3 Specification of the Pivot Interval Routing Scheme

We define a specification $SP_{PIR}$ for the Pivot Interval Routing Algorithm (PIR). We consider a computation $e$ of the PIR algorithm, to satisfy the specification $SP_{PIR}$ if the following conditions are true:

(SP1) $\forall i \in G, \exists j \in G : (name_i = name_j) \land i \neq j$ where $name_i$ and $name_j$ are the numbers assigned to the nodes $i$ and $j$ respectively in a graph $G$.

(SP2) $\forall i \in G, \forall j, k \in N_i, ((I(i,j) \cap I(i,k) = \Phi) \land (j \neq k))$

(SP3) A message traverses an optimal path if the source and destination are in the same ball.

(SP4) A message traverses a shortest distance (in the worst case) to the pivot that is nearest to the destination and the shortest distance from that pivot to the destination.

Note that SP1 guarantees that the arc labels assigned satisfying SP2, SP3 and SP4 aids in taking the message from the source to the destination as per the intention of the PIR algorithm.
CHAPTER 3

SELF-STABILIZING INTERVAL ROUTING ALGORITHM

In this chapter, we present a self-stabilizing pivot interval routing algorithm SPIR. Due to the complexity of the algorithm, we have attempted to divide the algorithm into several sub-algorithms. This algorithm SPIR is divided into the following sub-algorithms: i) Algorithm Min_BFS_trees, ii) Algorithm CalcSize_TreeClts, iii) Algorithm Conv_BcstSizes, iv) Algorithm Number_Nodes, v) Algorithm Label_Arcs and finally Algorithm Route_Msg_using_PIR. In the following section, we give the outline of the algorithm, introduce the data structures involved and then explain the operation of all the sub-algorithms.

3.1 Outline of Algorithm SPIR

In this algorithm, t-ball $B_u$ for each node $u$ is collected using some ordering $\prec$. The assumption that is made in this algorithm is that there is a self-stabilizing algorithm already running that collects the pivots of the graph $G$ and assigns pivot to each node, and at every node $u$ its pivot is represented by the variable $\text{pivof}_u$.

Algorithm SPIR has two phases: a node-labeling phase and an arc-labeling phase. Node-labeling is done as follows. Let $P = \{p_1, p_2, p_3, \ldots, p_t\}$ be the set containing all the
pivots of the graph $G$ where $l$ is the number of pivots and let $X_p = \{v | p(v) = p\}$ be its set of clients. For every $p \in P$, a minimum Breadth first tree (BFS) is constructed, rooted at $p$ and spanning the entire graph $G$, and let $T_p'$ be the sub tree of $T_p$ that contains only the clients of $p$. This is done in Algorithm Min_BFS_trees. All the minimum BFS trees are constructed in parallel for all the pivots. Then the size $S_z_p$ of $X_p$ in each $T_p$ is calculated in Algorithm CalcSize_TreeClts. Then, the labeling is performed in depth first manner by traversing the subtree $T_p'$ starting with the node number of root $p$ at $\sum_{i=1}^{p-1} S_z_p + 1$. This is done in Algorithm Number_Nodes.

The arc-labeling is done as follows. Let $All(G) = \{1, 2, \ldots, n\}$ be the set containing all the nodes in the graph $G$. Every node $u \in All(G)$, has a set of arcs $E_u$. Every arc $e \in E_u$ is labeled with a set of destinations $I(e) \subseteq All(G)$ such that every $e$ has a disjoint set $I(e)$ and also the collective $I(E_u) = All(G)$. The labeling is done in the following three steps. In the first step, in every tree $T_p'$, every node $u$ has its successors $s_1, s_2, \ldots, s_j$ where $j$ is the number of successors of $u$ and $j \geq 0$. Now every $I(u, s_k)$ where $k = 1, 2, \ldots, j$, is labeled with all the nodes in the subtree $T_{s_k}$ in $T_p'$. Let us denote all the nodes thus included in the collective $I(E_u)$ in this step as $M_1(u)$. In step 2, all the nodes in the set $M_2(u) = B_u(t) - M_1(u)$ are included in the set $I(E_u)$. For every node $v$ in the set $M_2(u)$, the shortest path from $u$ to $v$ is determined, and the arc $e \in E_u$ which is an arc on the shortest path to $v$ has $v$ included in its $I(e)$. Thus, steps 1 and 2 aid in taking a message through the shortest path to the destination if the destination is in the same ball as the source node. In step 3, all the nodes in the set $M_3(u) = All(G) - B_u(t)$ are included in the set $I(E_u)$. This is done as follows. If $e_p$ is an arc from the node $i$ to its predecessor in the tree $T_p$, then we add $I(e_p) =$
\[ I(e_p) \cup \{S_p \cap M_3(u)\} \] where \( S_p \) is the clients of \( T_p \). Step 3 helps the message to reach the destination’s pivot through the shortest path and from that pivot to the actual destination through the shortest path. We will see this in more detail in the sections to come.

3.2 Data Structures for Algorithm SPIR

We represent the parent of a particular node \( x \) in a given pivot tree \( T_p \) by \( par.x[p] \) where \( p \) is the pivot number and we denote the descendant relationship by \( D_x[j] \) where \( j \) ranges from 1 to \( k+1 \), \( k \) is the number of pivots in the graph \( G \). Every node \( x \) has a variable denoted by \( pivof_x \) which holds the name of the pivot to which node \( x \) is a client of, and another variable called \( Size_x \) which holds the size of the number of clients of its pivot in its sub-tree of that pivot tree. Every pivot node \( p \) has a variable called \( Sz_{Rdy_p} \) which is set to one whenever the number of clients in that pivot tree is found.

Some of the following algorithms have an underlying token circulation algorithm. So, in order to distinguish each token round, every node \( x \) maintains an array called \( C_x[j] \) where the number of elements of this array is \( k+1 \), \( k \) being the number of pivots.

Every node maintains another two variables called \( myhi_x \) and \( mylo_x \). The variable \( myhi_x \) maintains the highest node number in its sub-tree, consisting of its pivot’s clients, and the variable \( mylo_x \) maintains the node number of itself, that is node \( x \).

The variables \( ctree\_hi_x \) and \( ctree\_lo_x \) maintain the highest node number and the lowest node number respectively, in the whole sub-tree, with pivot as the root, consisting of the clients of the node.
This concludes the descriptions of the basic data structures used in the following sub-algorithms. There are a few more variables that are specific to some sub-algorithms, which will be introduced when we approach the corresponding sub-algorithms.

3.3 Algorithm SPIR in detail

3.3.1 Algorithm Min_BFS_trees

Multiple minimum BFS trees are constructed, one for every pivot $p$ with the pivot as the root. We used the algorithm proposed in [16] to build the minimum BFS trees.

3.3.2 Algorithm CalcSize_TreeClts

This algorithm uses an underlying token circulation algorithm [6] to calculate the number of clients of a pivot. To do this, the corresponding pivot tree $T_p$, where $p$ is the pivot number, is used. That is, this algorithm runs on every pivot tree with the different set of variables for each tree.

Informally, this algorithm can be explained as follows. The aim of this algorithm is to calculate the number of clients of every pivot. This algorithm is constructed in such a way that the circulating token visits only the clients of the corresponding pivot tree. The token for this algorithm is called Size-Token. When the Size-Token is passed on to an unvisited client node, the Size variable of its ancestor is set to one, and when all the descendants of a node in that pivot tree are visited, the Size variable of the ancestor of that node is incremented by the Size of that node. The round of token circulation starts
when the Size-Token is generated by the root node and is completed when the root node cannot pass on the Size-Token to any other node. At the end of the round, the root, which is a pivot, knows its number of clients and it is reflected by its Size variable.

Algorithm CalcSize_TreeClts is shown as Algorithm 3.3.2. Before we go into the details of the algorithm, we make the following observation:

**Observation 3.1** For every pivot $p \in P$, $T_p'$ is a connected tree.

**Proof:** $T_p'$ is the subtree of $T_p$ that contains all the clients of $p$. It is important to make this observation in order to calculate the correct size of the clients of $T_p$ in this Algorithm CalcSize_TreeClts. From the Definitions 2.3, 2.4 and 2.5, we see that every node $u$, assigns the closest pivot as its $pivof(u)$. So, for every node $v \in V$ and for every node $u \in V$ on a shortest path from $v$ to its pivot $pivof(v)$, $pivof(u) = pivof(v)$. Thus $T_p'$ is a connected tree. □

The macro $UV_i[k+1]$ represents the set of neighbors satisfying two properties: i) those that are not visited by the Size-Token and ii) those that are the clients of that pivot tree. $Search_i[k+1]$ chooses an element from $UV_i[k+1]$ using some ordering and this is the node to which the Size-Token is passed next.

A node is said to hold the Size-Token if the following predicate holds:

$$Size\_Token(i) \equiv Fwd\_CalcSz(i) \vee Btkk\_CalcSz(i)$$

$Fwd\_CalcSz(i)$ is enabled at the node $i$, when it receives a token for the first time from its parent, say, $A_i[k+1]$. On the other hand, $Btkk\_CalcSz(i)$ is enabled each time the token is backtracked to processor $i$ from its descendant $D_i[k+1]$. $Fwd\_CalcSz(i)$ and $Btkk\_CalcSz(i)$ are referred to as Forward($p$) and Backtrack($p$) in the token circulation algorithm. For a more detailed description of these predicates, refer to [6].
Algorithm 3.3.2 (CalcSize_TreeClts) Calculation of size of the clients of the pivot p using Depth-First Token Circulation.

Macro

\[
UV_i[k+1] = \begin{cases} 
q \in N \colon \left( (q \geq p, D_i[k+1]) \land (C_q[k+1] \neq C_i[k+1]) \land (D_q[k+1] \neq i) \land (\text{pivotof}_i = p) \right) & \\
\text{Search}_i[k+1] = \begin{cases} 
\min_i (UV_i[k+1]) & \text{if} \quad (UV_i[k+1] \neq \emptyset) \\
\bot & \text{Otherwise} 
\end{cases} & \\
Fwd_{\text{CalcSz}}(i) \rightarrow C_i[k+1] = (C_i[k+1]) \mod 2;
\text{Size}_i := \text{Size}_i + 1 \\
\text{if} \quad (i \in P) \quad S_z \_ \text{Rdy} := 1; & \\
Btk_{\text{CalcSz}}(i) \rightarrow \text{Size}_i := \text{Size}_i + \text{Size}_{D_i[k+1]}; \\
D_i[k+1] := \text{Search}_i[k+1]; & 
\end{cases}
\]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 3.1: Finding the size of the clients of the pivot $r$. 

- ○ Unvisited Node
- ● Visited Node
- ☒ Not a Client Node
Fwd_CalcSz(i) is enabled exactly once during each round, and this is guaranteed by the underlying token circulation protocol. When a node receives the token for the first time, it increments its Size variable by one. The node then passes the token to the next node which is a node that satisfies the descendant relationship, if any. Otherwise, it backtracks the token to its parent during which the parent updates its Size value to the sum of the previous value and the value of the descendant that just backtracked. From observation 1, the Size variable at the root p now contains the actual size of the clients.

Consider the example in Figure 3.1. In step ii), the root which is a pivot generates the Size_Token, makes its Size value 1. In step iii), the node b gets the Size_Token and makes its Size value as 1. In step iv), node c receives the token and sets its Size value as 1. In step v), the token is backtracked to b and b sets its value as 2. In step vi), the Size_Token is backtracked to node r and r updates its Size value as 3. In step vii), node a receives the token and it sets its Size value as 1. In step viii), the Size_Token is backtracked to the root r and the root updates its value as 4 and sets its Sz_Rdy variable to 1. Now that one round is completed, we see that the pivot holds the right size of its clients and the root is ready to start the next round.

3.3.3 Algorithm Conv_BestSz

Now that all the pivots know the size of its clients, they are now ready to share it with the rest of the pivots. This is done in this algorithm. This is done by converge casting all the sizes to the root which is a pivot and then broadcasting the collective set of the sizes of all the pivot clients to all the pivots, from the root. This can be done in one of the pivot trees and we have chosen the tree of pivot k to do this.
An informal description of this sub-algorithm is as follows. During the converge-cast phase, every node gets the sizes of the pivots in its sub tree from its descendants. Once this is done, it sends the sizes it collected to its ancestor. One round of converge-casting is said to be done when the root has the sizes of all the pivots.

Now the root is ready to broadcast during which sizes of all the pivots are broadcast to all the pivots. When this is done, another round of converge cast and broadcast follows.

The data structures involved in this algorithm are as follows. Every node $i$ maintains a variable called $Sz\_State_i$. $Sz\_State_i$ is an element of $\{I, C, B\}$ which stands for initial state, converge cast state and broadcast state respectively. The descendant relationship for this algorithm is denoted by $D\_Sz_i$. A variable with $k$ elements, where $k$ is the number of pivots, is maintained at every node, and is denoted by $PivSz_i[j]$. This variable is used to store the size of the pivots. $PivSz_i[j]$ contains the size of the clients of the $j$th pivot. Three set variables called $Set\_D_i$, $Rec\_D_i$ and $New_i$ are maintained. $Set\_D_i$ contains the set of all the descendants of $i$ in tree $T_k$. $Rec\_D_i$ contains the set of descendants from which the size information is not received by the node $i$. $New_i$ contains the pivot numbers, sizes of which are to be updated by its Ancestor in its $PivSz$ variable.

Algorithm $Conv\_BcstSz$s is shown as Algorithm 3.3.3. When the predicate $Upd\_Sz$s$(i)$ is invoked, the node updates its $PivSz$ variable with the sizes of the pivots available from the descendant that is ready to share and adds the pivot numbers that are updated to the $New_i$ set. It also removes the descendant that just shared the size information from the $Rec\_D_i$ set. Now if the $Rec\_D_i$ set is empty, then the $Sz\_State$ is set to $C$. The $Upd\_Sz$s$(i)$ predicate is invoked whenever a descendant becomes available to share the size
Algorithm 3.3.3 (Conv_BcstSzs) Broadcast of the sizes of all the pivots.

In Tree T_k.

Variable

\[ S_{\text{State}}_i \in \{I, C, B\} \]

\[ \text{PivS}_i[] = \{s_1, s_2, \ldots, s_k : s_j \in \text{integer}\} \]

\[ \text{Set}_D_i = \{q : q \in N_i \land (\text{par}.q[k] = i)\} \]

\[ \text{Rec}_D_i = \{q : q \subset \text{Set}_D_i\} \]

\[ \text{New}_i = \{q : q \in P\} \]

Predicate

\[ \text{Upd}_Szs(i) = \left( (S_{\text{State}}_D_{-s_i} = C) \land (D_{-sz_p} \in \text{Rec}_D_{-sz_i}) \lor (D_{-sz_i} = I) \right) \]

\[ \text{Upd}_ifPiv(i) = (\text{Rec}_D_i = \phi) \land (S_{\text{State}}_i = I) \land (i \in P) \land (S_{\text{Rdy}}_i = 1); \]

\[ \text{GetRdy}_Bcst(i) = (((S_{\text{State}}_i = C) \land (i \notin P)) \lor (i \in P)) \lor (S_{\text{State}}_i = C) \]

\[ \text{Do}_Bcst(i) = (S_{\text{State}}_i = B) \land ((S_{\text{State}}_i = I) \lor (i \in P)) \]

Actions

\[ \text{Upd}_Szs(i) \rightarrow \text{if} (D_{-sz_i} \neq I) \]

\[ \{ \forall j \in \text{New}_{D_{-sz_i}} (\text{PivSz}_i[j] = \text{PivSz}_{D_{-sz_i}}[j]); \text{New}_i := \text{New}_i \cup j; \} \]

\[ \text{Rec}_D_i := \text{Rec}_D_i - D_i; \text{if} (\text{Rec}_D_i = \phi) \text{Sz}_{\text{State}}_i := C; \]

else \text{Sz}_{\text{State}}_i := C; \]

\[ \text{Upd}_ifPiv(i) \rightarrow \text{PivSz}_i[i] := \text{Size}_i; \text{New}_i := \text{New}_i \cup i; \text{Sz}_{\text{State}}_i := C; \]

\[ \text{GetRdy}_Bcst(i) \rightarrow \text{Rec}_D_i := \text{Set}_D_i; \text{New}_i := \phi; \]

\[ \text{if} (i \in P) \{ \text{Sz}_{\text{Rdy}}_i := 0 \}

\[ \text{Sz}_{\text{State}}_i := B; \]

\[ \text{Do}_Bcst(i) \rightarrow \text{if} (i \notin P) \{ \text{PivSz}_i[] := \text{PivSz}_{A_i}[]; \}

\[ \text{Sz}_{\text{State}}_i := I; \]
Figure 3.2: Broadcasting the size collection of the pivots to all the pivots.
information it has. The $Upd_{IfPiv}(i)$ predicate is invoked when $i$ is a pivot and has the its $Sz_{Rdy}$ variable set to 1. It updates its own value of size of its clients, $Size$, in its $PivSz$ variable. $GetRdy_Bcst(i)$ is invoked when a node has all the sizes available. When this predicate is invoked, it sets its $Sz_{State}$ variable to $B$ and does some initializations for the next round of the Conv_BcstSzs algorithm. $Do_Bcst(i)$ is invoked when the ancestor signals a broadcast by setting its $Sz_{State}$ variable to $F$. When it's invoked, it copies the contents of the $PivSz$ variable of the Ancestor to itself and changes its $Sz_{State}$ to $F$.

At the end of one round of Conv_BcstSzs algorithm, all the pivots have the sizes of the clients of all the pivots and itself. Then another round of this algorithm is started. Figure 3.2 shows that the pivots have the sizes of all the pivots broadcast to them after one round of algorithm 3.3.3.

**3.3.4 Algorithm Number_Nodes**

Now that all the pivots know the sizes of previous pivots, the pivots are ready to initiate the node numbering in its own trees. This uses an underlying token circulation algorithm similar to the one in Algorithm 3.3.2. This algorithm runs in every pivot tree in parallel.

In this algorithm, all the clients in every pivot tree are numbered in the Depth first order. The numbering in each pivot tree starts at a number, which is the sum of the clients in the previous pivot trees plus one. Every node $i$ maintains a variable called $Step_i$. The variable $Step$ is set to 1 after the node has backtracked to its ancestor. This algorithm is constructed in such a way that the circulating token visits only the clients of the corresponding pivot tree as this tree is used to number only its clients. The token for this
algorithm is called Number_Token. When the Number_Token is passed on to an unvisited client node \( i \), the \( my_{lo} \) variable is set to \( my_{hi} \) variable of its ancestor \( A_i \) plus one. When the Number_Token is backtracked, \( my_{hi} \) variable of the ancestor of that node is incremented by the \( my_{hi} \) of this node. The round of token circulation starts when the Number_Token is generated by the root node and is completed when the root node cannot pass on the Number_Token to any other node. At the end of the round, all the clients in that tree have a number.

Formally, in this algorithm, Algorithm Number_Nodes is shown as Algorithm 3.3.4. The macros \( UV_i[p] \) and \( Search_i[p] \) respectively behave in a similar manner to that of the Algorithm 3.3.2.

The \( Fwd_{Num}(i) \) is enabled exactly once during each round and this is guaranteed by the underlying token circulation protocol. When a node receives the token for the first time, it sets its \( my_{lo} \) value as explained. The node then passes the token to the next node, which is a node that satisfies the descendant relationship, if any. Otherwise, it backtracks the token to its parent during which the parent updates its \( my_{hi} \) value to the sum of the previous value and the value of the descendant that just backtracked. The action for \( BkTk_{Num}(i) \) also includes one of the arc labeling step which is explained in the next section under Algorithm Label_Arcs.

**Claim 3.1** At the end of Algorithm Number_Nodes, every node \( x \) in the Graph \( G \) has a label \( \in \{1, 2, \ldots, n\} \) denoted by \( L(x) \) and for any node \( i \) and for any other node \( j \neq i \), \( L(i) \neq L(j) \).

Proof: From observation 1, since the tree \( T_p' \) is a connected tree, the Algorithm CalSize_TreeClts, after the round 1, has the size of its clients in its root \( p \). Since, in every
Algorithm 3.3.4 (Number_Nodes)  Labeling (Numbering) of clients of p using Depth-First Token Circulation.

In every tree $T_p$,

**Macro**

$$UV_i[p] = \left\{ q \in Ni : \left( (q \succeq p D_i[p]) \land (C_q[p] \neq C_i[p]) \land (D_q[p] \neq i) \land \left( ((C_q[p] \neq E) \lor (D_q[p] \neq \bot)) \land (\text{Pivot}_f = p) \right) \right\}$$

$$\text{Search}_i[p] = \begin{cases} \min \succ_i(UV_i[p]) & \text{if } (UV_i[p] \neq \emptyset) \\ \bot & \text{Otherwise} \end{cases}$$

**Predicate**

$$Fwd\_\text{Num}(i) = \begin{cases} \text{Forward}(i) \land (\text{Sz}_\text{Rdy}_i = 1) & \text{if } (i = p) \\ \text{Forward}(i) & \text{else} \end{cases}$$

**Actions**

$$Fwd\_\text{Num}(i) \rightarrow C_i[p] := (C_i[p] + 1) \mod 2;$$

$$\text{if } (i = p) \{ \text{if } (i = 1) \{ \text{my}_\text{lo}_i := 1; \} \}$$

$$\text{else my}_\text{lo}_i := \sum_{x=1}^{i-1} \text{Piv}\text{Sz}_i[x] + 1;$$

$$\text{else my}_\text{lo}_i := \text{my}_\text{lo}_i + 1;$$

$$\text{if } (D_i[p] = \bot) \{ \text{Step}_l_i := 1; \}$$

$$Bktk\_\text{Num}(i) \rightarrow \text{my}_\text{hi}_i := \text{my}_\text{hi}_{D_i};$$

$$I(i, D_i[p]) := I(i, D_i[p]) \cup \forall x(\text{my}_\text{lo}_{D_i[p]} \leq x \leq \text{my}_\text{hi}_{D_i[p]})$$

$$D_i[p] := \text{Search}_i[p];$$

$$\text{Step}_l_i := 1;$$
tree $T_p$, all the clients are numbered in the Algorithm Number.Nodes and every node is a client of exactly one tree, we conclude that all the nodes have a label at the end of the Algorithm Number.Nodes. At the end of the Algorithm Conv_BestSz, all the pivots know the sizes of all the other pivots in the graph. Then the predicate $Fwd_Num(i)$ is enabled at the root which starts numbering the root with $\sum_{x=1}^{i-1} PivSz_i[x]+1$. Thus the numbering of the clients in every tree is totally disjoint from the other trees. □.

3.3.5 Algorithm Label_Arcs

The arc-labeling algorithm shown in this section is constructed in three steps. The first and the second steps take care of enabling a shortest path delivery of the message to the destination if the destination is in the same ball as the source. If the destination is not in the same ball as the source, then the third step helps to take the message through a shortest path to one of the nodes in the destination’s ball, that is either the destination’s pivot or some other node that’s on the way to the pivot and is closer to the destination than the pivot.

The aim of this algorithm is to do the arc labeling, enabling every message to reach the right destination using an optimal path if the source and the destination are in the same ball, and using a near-optimal path if the source and the destination are in different balls. Every node $i$ maintains a set variable called $I(i, x)$ ($x \in N_i$) for every neighbor of $i$. The first step of arc labeling enables the message to travel the optimal path from the destination’s pivot to the destination. This is taken care of in the previous section in the Algorithm Number.Nodes. In that algorithm, $Bulk_Num(i)$ is invoked when the
Number_Token is backtracked to the ancestor \( i \) from a descendant \( D_l[p] \) in the tree of pivot \( p \). When \( \text{Btk}_\text{Num}(i) \) is invoked, all the numbers between and including \( \text{my}_\text{hi}D_l \) and \( \text{my}_\text{lo}D_l \) are added to the set \( I(i, D_l) \). Now the message from the pivot to its client can be reached via the optimal path by following the arc labels.

Algorithm Label_arcs is shown as two algorithms to show steps 2 and 3 separately. Let us call them Algorithm Label_arc_step2 and Algorithm Label_arc_step3 respectively. Step 2 is done after Step 1 is done. Algorithm Label_arc_step2 knows that Step 1 is done by looking at the variable Step_1. If it is equal to 1, then this algorithm is initiated. This algorithm depends on the underlying minimum spanning trees built with every node as the root and spanning only the nodes that are in the ball. Let us call these trees \( \text{small}_\text{tree}_i \) where \( i \) is the number of the node. Every node \( x \) associated with a \( \text{small}_\text{tree}_i \) has variables \( \text{par}_\text{in}_\text{stree}_x[i] \) which has its parent in the small tree stored in it, and a set variable \( \text{col}_\text{intls}_x[i] \) which collects the intervals from its descendants that is to be added to the arc of the neighbor of \( i \) and is on the path to \( x \). The descendant relationship for this step of arc labeling is represented by \( D_l[i] \).

The arc labels of \( i \) should include all the node numbers in its \( \text{small}_\text{tree} \) so that a message travels the shortest distance to every node in the \( \text{small}_\text{tree} \) from \( i \). To do this, a converge cast is done in all the \( \text{small}_\text{trees} \) from every leaf node so that \( i \) gets all the node numbers leading to that leaf node.

Algorithm Label_arc_step2 is shown as Algorithm 3.3.6. During the converge cast phase, every node gets all the node numbers in its subtree from its descendants and stores it in the \( \text{col}_\text{intls} \) variable. Once this is done, it shares that variable with its ancestor. One
Algorithm 3.3.6 (Label_Arcs_Step2) Labeling of specific arcs using shortest path spanning tree algorithm.

**Variable**

$I_{\text{State}} \in \{S, D\}$

$PivS_z[] = \{S_1, S_2, \cdots, S_k : S_j \in \text{Integer}\}$

$Set_{\text{L}} = \{q : (q \in N_i) \land \text{par.q[k]} = i;\}$

$Rec_{\text{L}_i} = \{q : q \in Set_{\text{L}_i}\}$

$Col_{\text{Intls}} \subset \{1, 2, \ldots, n\}$

**Predicate**

$Upd_{\text{Intls}}(i) \equiv (\left(\left(I_{\text{State_{D-i}}} = D\right) \land (D_{\text{l}_{i}} \in Rec_{\text{L}_i})\right) \lor (D_{\text{l}_{i}} = \top)) \land (I_{\text{State}} = S) \land (\text{Step1}_{i} = 1)$

**Actions**

$Upd_{\text{Intls}}(i) \rightarrow \text{if}(D_{\text{l}_{i}} \neq \top)$

\{\text{Col}_{\text{Intls}_{i}} := \text{Col}_{\text{Intls}_{i}} \cup \text{Col}_{\text{Intls}_{D_{\text{l}_{i}}}};$

\text{if}(i \in j)\{

$\forall x \in \text{Col}_{\text{Intls}_{i}} \{\text{if}(\exists y : x \in I(i, y))$

$I(i, D_{\text{l}_{i}}) = I(i, D_{\text{l}_{i}}) \cup x;\}$

$Rec_{\text{L}_i} := Rec_{\text{L}_i} - D_{\text{l}_{i}};$

$\text{if}(Rec_{\text{L}_i} = \phi)\{$

$I_{\text{State}} := \top;$

$I_{\text{State_{D_{\text{l}_{i}}}}} := S;$

$Rec_{\text{L}_i} := Set_{\text{L}_i};$

$\}$

$\}$

$\text{else } I_{\text{State}} := \top;$

$\text{Step2}_{i} := 1; \text{Step1}_{i} := 0;$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 3.4: Labeling the arcs to travel the shortest path inside the ball.
round of converge cast is said to be done when the root has all the node numbers in its small_tree.

Every node $i$ maintains a variable called $I_{State_i}$. $I_{State_i}$ is an element of $\{S, D\}$ which stands for ready to start converge cast, and converge cast done respectively. As the root receives the node numbers from a descendant, it includes those numbers in the $I$ variable corresponding to that descendant if its not included in any other arc in Step 1 of the arc-labeling algorithm. Once the converge cast is done, the root of the corresponding small_tree sets its Step2 variable to one. Figure 3.4 shows the interval set at each node before and after one round of Algorithm 3.3.6. Once step 2 is done, the nodes are ready to start step 3. Algorithm Label_arc_step_3, again uses an underlying token circulation algorithm. This algorithm runs on every pivot tree.

Formally, in this algorithm, we denote the descendant relationship of a node $i$ by a variable $D_i[p+k+1] \ (D_i[p+k+1] \in N_i \cup \bot)$ where $p$ is the pivot number and $k$ is the number of pivots. To distinguish each token round, each node maintains a variable $C_i[p+k+1]$ and this is called the round color for this algorithm which is denoted as $C_p$ in the token circulation algorithm.

Algorithm Label_Arc_Step_3 is shown as Algorithm 3.3.7. The macros $UV_{I_i[p+k+1]}$ and $Search_{I_i[p+k+1]}$ behave like the macros UV and Search in Algorithm 3.3.2 but the token here is $I_{Token}$. The predicate $Fwd_{Label}(i)$ accommodates an extra condition to check if Step 2 was completed before starting Step 3 of arc labeling.

Every node $i$ maintains the variables called $ctree_lo_i$ and $ctree_hi_i$ which are used to store the highest number of the clients in that pivot tree. The token for this algorithm is
Algorithm 3.3.7 (Label_Arcs_Step3)  Labeling of remaining arcs using Depth_First Token Circulation algorithm.

Macro

\[
UV_i[p + k + 1] = \begin{cases} 
q \in N_i : \left( q \Rightarrow D_i[p + k + 1] \land (C_q[p + k + 1] \neq C_i[p + k + 1]) \right) \\
\neg (D_q[p + k + 1] \neq \bot) \land (par_q[p] = i) 
\end{cases}
\]

\[
Search_i[p + k + 1] = \begin{cases} 
\min \Rightarrow (UV_i[p + k + 1]) & \text{if } (UV_i[p + k + 1] \neq \phi) \\
\bot & \text{otherwise} 
\end{cases}
\]

Predicate

\[
Fwd\_label(i) = Forward(i) \land (Step_2 = 1);
\]

Actions

\[
Fwd\_label(i) \rightarrow C_i[p + k + 1] := (C_i[p + k + 1] + 1) \mod 2;
\]

\[
\begin{align*}
&\text{if}(i = p)\{ \\
&\text{ctree}_hi_i := my\_hi_i; \\
&\text{ctree}_lo_i := my\_lo_i;\}
\end{align*}
\]

\[
\begin{align*}
&\text{else}\{ \text{ctree}_hi_i := \text{ctree}_hi_A; \\
&\text{ctree}_lo_i := \text{ctree}_lo_A;\}
\end{align*}
\]

\[
Btkk\_label(i) \rightarrow \forall j(\text{ctree}_lo_i \leq j \leq \text{ctree}_hi_i)
\]

\[
\begin{align*}
&\{ \\
&\text{if}(\exists j \in I(i, N_i))
\end{align*}
\]

\[
\begin{align*}
&I(i, D_i[p + k + 1]) := I(i, D_i[p + k + 1]) \cup \\
&(\forall x : my\_lo_{D_i[p]} \leq x \leq my\_hi_{D_i[p + k + 1]})
\end{align*}
\]

\[
D_i[p + k + 1] := Search_i[p + k + 1];
\]

\[
Step_2_i = 0;
\]
Figure 3.5: Labeling the arcs to travel the shortest possible path between the balls
called I-Token. When the I-Token is passed on to an unvisited client node \( i \), it updates its \( ctree_{lo} \) and \( ctree_{hi} \) variables from its ancestor \( A_i \). When the I-Token is backtracked to node \( i \), all the numbers including and between \( ctree_{lo} \) and \( ctree_{hi} \) are added to the variable \( I(D[p+k+1], i) \) if they are not added to any of the I variables of \( D_i[p+k+1] \) during the steps 1 and 2 of the arc-labeling scheme. At the end of the I-Token circulation round, arc-labeling is complete.

**Claim 3.2**  
For every node \( x \in V \), \( \{I(e) | e \in E_x\} \cup L(x) = \{1, 2, \ldots, n\} \).

**Proof:** In Step 1 of the arc-labeling scheme done in Algorithm Number_Nodes, whenever the predicate \( BkTk_Num(i) \) is enabled, it labels its corresponding arc through which it is backtracking with the set of destinations that are the successors as explained in the outline of the Algorithm SPIR. So at the end of Step 1, every node \( u \) has its arcs \( I(E_u) \) labeled with \( M_1(u) \). In Step 2, the set \( M_2(u) \) which is \( B_u(t) - M_1(u) \) is added to the set \( I(E_u) \). This follows from the Algorithm Label_Arcs_STEP2. Then finally, in the Algorithm Label_Arcs_STEP2, all the other nodes that are not in the set \( I(E_u) \) are included to the set. Since every node belongs to a unique set \( S_p \) for every \( p \in P \), Step 3 of the arc-labeling algorithm guarantees that every node will be contained in one of the sets \( I(e) \).

□

**Claim 3.3**  
For every two distinct arcs \( (x,y) \) and \( (x,z) \), \( I(x,y) \cap I(x,z) = \emptyset \).

**Proof:** In every step of the arc-labeling algorithm, we exclude the nodes that are already included in the set \( I(E_u) \) and attempt to label every arc with only the nodes that are not a part of \( I(E_u) \). In Step 3, we rely on the fact that \( \{S_p\}_P \) is a partition of \( V \). So, it’s clear that in every step, a label is inserted only into one of the sets \( I(e) \).

□
3.3.6 The Routing Module

The routing module for SPIR is very simple and is as shown as below:

Receive msg,(l,msg)
If (l = i)
   deliver _msg;
else
   if (∃I(i,x): l ∈ I(i,x))
   Send msg_i to output _port(x);

Let us consider an example to explain the routing module. Say, a node i wishes to send a message to a node j. Node i checks the intervals to find which interval has the node j in it. Say I(i, t) has the node j in it. Then node i sends the message to the port which goes to the node t. When t receives the message, if t is the destination, then the message is delivered else it is again forwarded as done by node i. This continues till the message reaches the destination j.
CHAPTER 4

PROOFS OF CORRECTNESS FOR THE SPIR ALGORITHM

Node numbering phase comprises Algorithm 3.3.1, Algorithm 3.3.2, Algorithm 3.3.3 and Algorithm 3.3.4. Arc labeling phase comprises Algorithm 3.3.6, Algorithm 3.3.7 and a part of Algorithm 3.3.4.

Let us assume that the legitimacy predicates of the shortest path spanning tree algorithm used in Algorithms 3.3.1 and 3.3.6 as \( L_{SO} \) and the depth-first token circulation algorithm used in several algorithms as \( L_{TC} \). Now we define the legitimacy predicates, \( L_{NL} \) and \( L_{AL} \) for the node labeling and arc labeling phases, respectively, as follows:

\[
L_{NL} \equiv L_{SO} \land L_{TC} \land SP1 \\
L_{AL} \equiv L_{NL} \land SP2 \land SP3 \land SP4
\]

In section 4.1, we show the correctness proofs for the node-labeling phase and in section 4.2, we show the correctness proofs for the arc-labeling phase and finally in section 4.3, we show the complexity analysis for the SPIR algorithm.

4.1 Correctness of the Node Numbering Phase

**Theorem 4.1** \((L_{SO} \land L_{TC}) \Rightarrow L_{NL}\)

Proof: Closure: Follows from Algorithm SPIR.

Convergence : Follows from Claim 3.1.
4.2 Correctness of the Arc Labeling Phase

**Lemma 4.2** At every node the arc labels are totally disjoint and include all the nodes in the graph except itself in exactly one of the intervals.

Proof: Follows from the Claims 3.2 and 3.3

□

**Lemma 4.3** For every two nodes, \( u, v \in V \), there exists a sequence of nodes \( u = x_i, x_{i+1}, ..., x_l = v \) such that \( L(v) \in I(x_i, x_l) \) for every \( 2 \leq l \leq l \) (namely every message arrives at its destination).

Proof: Let us set the priorities of a message \( M \) based on the destination. If the destination is included in the source’s \( I(e) \) using Step1 of the arc-labeling scheme, then the priority of the message \( M \) is 1. Else its priority is 2 or 3 depending on whether it is included in the set \( I(e) \) in Step 2 or 3. Let us prove this lemma by showing that the sequence of priorities of a message on the path traversed by the message is non-increasing. If the priority of a message is 1, then eventually the destination is reached, and if the priority of the message is 2 or 3, eventually the destination is reached or the priority of the message decreases.

Consider a message \( M \) with source \( u \) and destination \( v \). Assume that its priority is 1. Then the Message goes to one of its successor \( x \) that is strictly closer to \( u \) than \( v \). And we see that \( x \) again has priority 1 for the message \( M \). So, by induction and Claim 3.3, we can conclude that the message \( M \) eventually reaches the destination \( v \).
Assume that the priority of $M$ is 2. In this case, as the message approaches the destination $v$, the priority remains 2 for some time and then it decreases to 1 and we see that it eventually reaches the destination.

Now, let us assume that the priority of the message $M$ is 3. Now, till the message reaches the ball of the destination pivot’s ball $B_p(t)$, we see that its priority remains 3. Once it reaches $B_p(t)$, its priority decreases to either 2 or 3 and we see that it eventually reaches the destination $v$. □

**Lemma 4.4** The arcs are labeled in such a manner that if the source and the destination are not in the same ball, a message traverse the shortest path from the source to a node in the ball of the destination and from that node it traverses the shortest path to the destination.

Proof: Follows from the Algorithms 3.3.4, 3.3.7. and lemma 4.3. □

**Theorem 4.2** $L_{NL} \triangleright L_{AL}$

Proof: Closure: Follows from Algorithm SPIR.

Convergence: Follows from Theorem 4.1 and Lemmas 4.2, 4.3, 4.4.

**Theorem 4.3** Algorithm SPIR is self-stabilizing.

### 4.3 Complexity Analysis

The space and time complexity for Algorithm SPIR is as follows. The space complexity represents the amount of space required to hold the interval information. The number of intervals per arc is at most $3n^{1/2}(1+\log n)^{1/2}$. The amount of bits required to store an interval is $2\log n$. So at each node, if the number of arcs is $m$, the space complexity is $6mn^{1/2}(1+\log n)^{1/2}\log n$. So the overall space complexity for the whole
network of \( n \) nodes is \( O(n^{3/2}\log n^{3/2}) \) bits. The time taken for SPIR to stabilize after the underlying depth-first token circulation and the shortest path spanning tree algorithms is \( O(n) \) steps.

SPIR algorithm has an average stretch factor of 3 as proposed by [24]. This claim can be established by showing that for every two nodes \( u, v \in V \),

\[
\frac{\text{dist}(\text{SPIR}, u, v)}{\text{dist}(u, v)} + \frac{\text{dist}(R, v, u)}{\text{dist}(v, u)} \leq 6
\]

where \( \text{dist}(\text{SPIR}, i, j) \) is the distance traversed by a message from node \( i \) to node \( j \) using the SPIR algorithm and \( \text{dist}(i, j) \) is the distance traversed by a message from node \( i \) to node \( j \) using the shortest path possible.

We consider three cases:

Case 1: \( u \in B_v(t) \) and \( v \in B_u(t) \). Then a message from \( v \) to \( u \) and a message from \( u \) to \( v \) will both traverse a shortest path. Thus, a sum of the stretch factors of both paths is 2.

Case 2: \( u \in B_v(t) \) but \( v \notin B_u(t) \) (or vice-versa). Then a message from \( v \) to \( u \) will traverse a shortest path, and the message from \( u \) to \( v \) will traverse a path of length at most \( 5\text{dist}(u, v) \) [24]. Thus the sum of the stretch factors is at most 6.

Case 3: \( u \notin B_v(t) \) and \( v \notin B_u(t) \). We bound the stretch factor of the path of a message from, say, \( u \) to \( v \), and the same bound holds symmetrically for the path of a message from \( v \) to \( u \). Since \( u \notin B_v(t) \), and pivot of \( v \), \( p(v) \), is an element of \( B_v(t) \), we have that \( \text{dist}(p(v), v) \leq \text{dist}(u, v) \). It follows that \( \text{dist}(u, p(v)) \leq \text{dist}(u, v) + \text{dist}(v, p(v)) \leq 2\text{dist}(u, v) \).

Thus, \( \text{dist}(\text{SPIR}, u, v) \leq \text{dist}(u, p(v)) + \text{dist}(v, p(v)) \leq 3\text{dist}(u, v) \). Symmetrically,
\(\text{dist}(\text{SPIR}, v, u) \leq 3\text{dist}(u,v)\). Thus, in this case as well, the sum of the stretch factors of the path of a message from \(u\) to \(v\) and from \(v\) to \(u\) is at most 6. It follows that

\[
\text{Average Stretch Factor}(\text{SPIR}) = \frac{1}{n(n-1)} \sum_{uv} \frac{\text{dist}(\text{SPIR}, u, v)}{\text{dist}(u, v)}
\]

\[
= \frac{1}{n(n-1)} \sum_{uv} \left( \frac{\text{dist}(\text{SPIR}, u, v)}{\text{dist}(u, v)} + \frac{\text{dist}(\text{SPIR}, v, u)}{\text{dist}(v, u)} \right)
\]

\[
= \frac{1}{n(n-1)} \sum_{uv} 6 \leq 3.
\]
CHAPTER 5

CONCLUSION

We presented a self-stabilizing interval routing scheme SPIR. In high-speed networks, the bottleneck is the delay in the nodes and not the propagation time of the messages. As high-speed networks gain popularity, it is essential to design direct routing schemes, which could be implemented in hardware and at the same time, when dealing with large networks, it is important to decrease the amount of memory kept in the nodes for routing purposes. In this work, we present for the first time a self-stabilizing routing strategy that generates for every unweighted network, a direct routing scheme with small stretch factor and modest memory requirements. This algorithm works on any anonymous, asynchronous and arbitrary unweighted networks. These can be easily extended to the weighted networks also. This algorithm takes $O(n)$ steps to stabilize once the underlying token-circulation and shortest path spanning-tree algorithms are stabilized. SPIR has memory requirement of $6mn^{1/2}(1+\log n)^{1/2}\log n$ at every node and $O(n^{1/2}\log n^{1/2})$ bits in total.

This algorithm is a good step forward in the research of compact routing protocols because it achieves near-optimality with a very simple algorithm compared to the hierarchical routing algorithms. Still the stretch factor and the memory requirements remain almost the same as the hierarchical routing methods. The stretch factor has almost
reached the lower bound of the compact routing protocols. But how about the memory requirements? Is there any way to still improve it while maintaining the virtues of this algorithm? Or, is it possible to improve the worst case stretch factor of 3 using the same amount of memory? Also, what is the best trade-off between the memory requirements and the average stretch factor for routing strategies that are hierarchical in nature? We leave these questions open for the future area of research.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Sripriya Sundaram

Local Address:
2066 Fruitdale Ave #12
San Jose, CA 95128

Home Address:
10/2 Kalaimagal Nagar
Ekkadutangal, Chennai, India 600097

Degrees:
Bachelor of Engineering, 1996
University of Madras

Thesis Title: Self-stabilizing Interval Routing Algorithm With Low Stretch factor

Thesis Committee:
Chairperson, Dr. Ajoy Kumar Datta, Ph.D.
Committee member, Dr. John T. Minor, Ph.D.
Committee member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Lori Mann Bruce, Ph.D.