Self-stabilizing routing protocols

Rajesh Viswanathan

University of Nevada, Las Vegas

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

Repository Citation

https://digitalscholarship.unlv.edu/rtds/396

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

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
Self-stabilizing routing protocols

Viswanathan, Rajesh, M.S.
University of Nevada, Las Vegas, 1994
The thesis of Rajesh Viswanathan for the degree of Master of Science in Computer Science is approved

Chairperson, Ajoy Kumar Datta, Ph.D

Examining Committee Member, Kazem Taghva, Ph.D

Examining Committee Member, Laxmi Gewali, Ph.D

Graduate Faculty Representative, Ashok Iyer, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas
August 1994
ABSTRACT

In systems made up of processors and links connecting the processors, the global state of the system is defined by the local variables of the individual processors. The set of global states can be defined as being either legal or illegal. A self-stabilizing system is one that forces a system from an illegal state to a global legal state without external interference, using a finite number of steps. This thesis will concentrate on application of self-stabilization to routing problems, in particular path identification, connectivity and methods involved in destinalational routing. Traditional methods for creation of rooted paths to multiple destinations in a computer network involve the creation of spanning trees, and broadcasting information on the tree to be picked up by the individual nodes on the tree. The information for the creation of the tree are all sourced at the root, and the individual nodes update information from the centralized source. The self-stabilization model for networks allows the decision for a creation of a tree and message checking to occur automatically, locally, and more important, in contrast to traditional networks, asynchronously. The creation, "message" passing occur with a node and its immediate neighbor, and the tree, path is created based on this communicated data. In addition, the self-stabilization model eliminates the requisite initialization of traditional networks, i.e. given any arbitrary initial state the system (a given network) is guaranteed to stabilize to a legal global state, in the case of a broadcast network, a minimal spanning tree rooted at a source.
ACKNOWLEDGEMENTS

I would like to thank Dr. Datta for his extreme patience, and guidance. Special acknowledgement to everyone in the Department of Computer Science, with particular thanks to Dr. Yfantis, and Dr. Gewali for their support. I would also like to express my gratitude to Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Ashok Iyer for serving on my committee. Last, but not least, endearing gratitude to my family for emphasizing the virtue of knowledge.
Chapter 1

Introduction

An asynchronous distributed system is a system of loosely coupled state machines that do not share a common memory. Formally we define a graph (or network or system) of being composed of a set of nodes, and a set of edges that connect pairs of nodes. Edges have a cost associated with them. In such a system, each machine, (node), has a local state. Every node has access to its neighbors local state (normally accomplished by exchanging messages) to determine its local state at a given time. In a self-stabilizing system, it is assumed that the local variables, whose content determines local state of a node can be read by its neighboring nodes, which constitutes the message passing model in distributed systems. The set of all local states yields a global state, which may be defined as (1) legal or (2) illegal depending on the definition of the global state and adherence to the description by the local states. Initially when the system is started, the local states are arbitrary, and thus the global state may be illegal. When in this illegal state, a node or many nodes may have a privilege, i.e., a move or step defined to correct the illegal state defined by a set of rules, which make up the self-stabilization algorithm.

This model was originally proposed by Dijkstra [1] for a ring of finite-state machines and used for the mutual exclusion problem. More recently, this model has been used to solve a variety of other problems, including leader election,
network protocols, breadth-first-search, spanning tree, and maximal matching problems. Self-stabilizing algorithms do not need to be initiated when a perturbation occurs without any external intervention, since they run continuously. The algorithm will require a node to make a move, i.e., change its state, when it is privileged, i.e., in a state where it is able to make a move. After a finite series of privileges and moves, the system will converge to a global legal state within a finite time, after which no more privileges will be created.

This research concentrates on the application of self-stabilization to the area of computer networks, and especially in the area of algorithms related to the detection of a path to a destination, and routing to a destination. Noteworthy is the effort of self-stabilizing systems already use in practical network routing protocols like the IGRP (Interior Gateway Routing Protocol), and OSPF (Open Shortest Path First). Previous network protocols used as practical applications, and those studied have been limited by their inability to take into consequence the dynamic nature of a group of computers linked together. Particularly unappealing is the synchronous, centralized nature of the algorithms, detrimental to the fast, adaptive advantage offered by computer networks. This research introduces a pragmatic approach for the problem of routing, and algorithms associated with routing, taking into account the inherent dynamic nature of self-stabilization. The essence of self-stabilization lies in asynchronous, distributed solution the individual nodes achieve, irrespective of the initial states each the nodes.

1.1 Definition of Self-Stabilization

The idea of self-stabilization was conceptualized by Dijkstra [1]. Dijkstra's model, originally proposed for one particular problem, has found great acceptance and recognition in areas far ranging, including real-time systems, communication
protocols, and any area relevant to fault-tolerance. The notion of self-stabilization has offered an opportunity to approach fault-tolerance including transient failures in distributed systems, in a formal manner. Dijkstra defined the concept of a privilege, whereby a node finding itself in a globally defined illegal state, can move to a state different from its current value. The definition of self-stabilization is thus derived to be when a system, regardless of its initial state, and the moves made by the individual nodes in the system, is guaranteed to reach a legal configuration within a finite number of moves. For any system self-stabilization is defined to be legal (legitimate) state if the following properties are satisfied:

[P1] Starting from an arbitrary state, the system will converge to a global legal state within a finite number of state moves.

[P2] Once the system is in the global legal state, there are no more privileges in the system, and thus the system is said to be in deadlock.

A transient failure is an event that may change the state of a system. This change may result in changing the legal global state of a system, as a consequence of corrupting the memory channels, communication channels, process failures and recoveries, etc. Self-stabilization models work under the premise that the system is able to recover from these transient failures, assuming they do not occur forever. The correctness of the algorithm is thus proven by showing that they satisfy properties P1, and P2. In this research the proof of correctness is shown through the graph theory arguments rather than the method of rounds presented by Dolev [3]. The communication analysis of such a system is complicated without additional systems. Though the concept of round, and cycles are not applicable to a distributed, asynchronous model, they can be used to compute the communication complexity given the definition of a round and a cycle as in [3]. Each cycle is
defined to include a series of steps in which (a) a node receives some variable value from one of its neighbors, and (b) copies the value into its own variable. The value copied could be a result of any privilege the particular node acquires, and is not restricted to any one move at any cycle. A round is defined complete when every node has executed exactly one cycle. Thus a round that has completed is defined by a set of nodes that had privileges during that round completing their independent moves. Alternate proof methods also include defining a bounded function, and showing the convergence of the parameters in the algorithm within their bounded values. However, it is often difficult to define a single bounded function for the algorithm, and other complex solutions have to be added.

The application of self-stabilization has transcended the initial area of intent and into fields as diverse as real-time transactions to communication protocols. The idea of system stabilizing in spite of incorrect initialization, and transient failures has attracted a broad range of applications suited for self-stabilization. Self-stabilization extends itself very well to network related problems, given the inherent need for a computer network to keep abreast of all changes in the system so at any given time a accurate view of the network is maintained by all necessary components. There has been much recent interest in both the practical and research world into this exact application, and continued study into the area of self-stabilization as a whole.

1.2 Routing Model

For our model we consider a system as comprising a set of nodes connected together in an arbitrary topology. The topology of the system can be defined as a connection of the components of the system by edges, either directed or undirected. Each node of the system has a local state, and a global state is defined, applicable to
all the system components, and change in the local states is with respect to their reference to the global state. In any practical network, the real function is the routing of information or data between a source node to the destination node. In most networks, the information requires travel through multiple number of system components. Routing algorithms are a major component of the network layer design, and choose the routes and data structures that a certain network architecture uses. Along with sheer mass of the data necessary to route packets to a large number of networks, there are many problems with the updating, stability, and cost minimization of the routing algorithms. Much research is being done in the area, but the optimal solution to these routing problems is still years away. In most cases, the routing we have today works, but sub-optimally and sometimes unpredictably. For an idea of the nature of real-life networks, a little background is necessary. Real networks have designated gateways or routers which connect several networks together, and pass information between them. The decision on what data to send in between the different nets is based on the information included in the information packet, called the header, and the state of the networks itself. The header primarily contains the destination address, a unique id designated to each component of the large system. The state of the network, that is links or nodes which might be up or down at any given moment, is determined by these gateways passing information among themselves. The distribution of the database (what each node knows), the form of the updates, and the metrics used to measure the value of a connection, are the parameters which determine the characteristics of a routing protocol.

Under some routing methods, each component in the graph has complete knowledge of the state of the network. This implies that these nodes must contain large amounts of local storage, and processing power to search through large tables in short time (which has to be done for each packet of data flowing through a node). The nodes also has take into consideration changes affecting the current
situation in the network. There are several problems with this type of approach, one of which is to produce a closed loop and infinite circulation of data around the loop, until it is discarded (due to a field in the data packet called TTL or time to live. The alternatives are thus to seek distributed algorithms which respond to dynamically to the network parameters, and update their local information restricted to a small subset of the network, in the case of a self-stabilizing algorithm, a node's immediate neighbors. There are several advantages offered by self-stabilizing routing algorithms:

- The algorithm runs continuously (there is no initiation of the algorithm necessary)
- Any failed link or node automatically re-evaluates the spanning tree relationship, and subsequently the broadcast message ids.
- There is no initialization of the local variables, because a self-stabilizing algorithm does not require any initialization.
- The nodes can move be any order, and the system will still stabilize.

The research work presented here also applies earlier work done in self-stabilizing algorithms essential for routing models. The combination suggests a layering of one or more algorithms, and is used frequently in self-stabilizing problems, including solutions to stabilizing graph coloring [9].

1.3 Notations and Demons

A self-stabilization algorithm is expressed in terms of guards, and privileges. The programs follow the notation:

<statement>
<statement>

...
with each statement having the form: `<guard> → <action>`

A guard is a boolean expression over the variables that a machine can read (its own along with any of its neighbors, i.e., adjacent processes). If some machine has a statement whose guard is true, then that machine has a privilege and may make a move. No assumptions are made about this choice. It assumed in this research that any node with a privilege can, and will make a move in finite time, and that more than one node can make a move at a time (i.e., a distributed demon is assumed [1,8]). There are four schedulers defined, the simplest of the four being centralized demon, which Dijkstra assumed for the original paper:

- central demon: Moves are executed atomically, one at a time. A central demon selects a move from a set of possible privileged processes and executes the privilege.
- randomized central demon: Moves are executed atomically, one at a time. The choice of the move to be made is conducted in a random fashion.
- distributed demon: Multiple moves by privileged nodes are allowed. In the presence of a distributed demon any subset of the set of privileged nodes can make a move at any point in time.
- read/write demon: Communication is through shared registers, with the all shared registers serialized according to read/write operations.

The algorithms presented here work in the presence of the distributed demons.
Chapter 2

Broadcast Routing

Flooding is an extreme method of routing. It uses the relatively simple idea of forcing every incoming message to leave out through every other channel, except the channel(s) the data was received through. The obvious problem is the number of wasteful retransmissions, and increment in the network traffic. There are two solutions to reduce the number of retransmissions, one is to include a "hop" counter, which gets every time a node processes the information, and retransmits until the diameter (i.e. the longest continuous path) is reached, at which point the information is ignored. Another method used to reduce the high traffic is to include the source identity of each information received by a node, and discarding duplicate copies of the data from the source. A not-so obvious advantage offered by this method is the property of flooding to guarantee delivery of data to a destination, given all possible routes between a source and destination are not corrupt. This high degree of robustness is desirable in any network architecture. It is for this very reason, flooding is used in certain network architectures as a method to initiate a path between a source and destination before the actual transmission of information. Similar to flooding, broadcasting is another way of distributing the data from a source to all the components in the system. Like flooding, broadcasting is used as a way to disseminate information throughout the graph, however, in contrast to flooding, reduces the traffic by decreasing the number of transmissions in the
network. The use of broadcasts, especially on high-speed local area networks, is a good base for many applications. Broadcasts are useful when a host needs to find information without knowing exactly what other host can supply it, or when a host wants to provide information to a large set of hosts in a timely manner. When a host needs information that one or more of its neighbors might have, it could have a list of neighbors to ask, or it could poll all of its possible neighbors until one responds. Use of a wired-in list creates obvious network management problems (early binding is inflexible). On the other hand, asking all of one’s neighbors is slow if one must generate plausible node ids, and try them until one works.

One way of routing using broadcasting is to build a spanning tree. Thus each component in the system is aware of which of its channels are part of the spanning tree and can restrict the transmission of data through those edges alone. This method of routing establishes a much more efficient use of the underlying network than does flooding, generating the absolute minimum number of retransmissions, in the absence of failures, necessary to complete the job.

2.1 Spanning Tree

Our study of spanning trees is restricted to their immediate application to broadcast routing. Spanning trees have been studied extensively in the past, and a number of algorithms have been proposed and analyzed for constructing spanning trees. There are well-established sequential algorithms based on the work of Kruskal [11] and Prim [11], as well as proposals for distributed versions of spanning tree construction [12]. The emphasis of this research is the application of minimal spanning trees to broadcast routing. In practical network applications, like ARPANET, delay estimates to all computer nodes connected in a network are performed by establishing a minimal spanning tree and routed to all the nodes by a
broadcast. Given our definition of a graph as a set of edges, connecting nodes, a spanning tree is defined as a subset of the edges such that there is a path between every pair of nodes, without a cycle. A minimal spanning tree for such a network is a spanning tree for which the total of the cost of the channels is minimal. The study of self-stabilizing spanning tree constructions has been studied, and algorithms established [5]. This research will extend the algorithms to the encompass the construction of minimal spanning trees, and layer the self-stabilization of broadcast routing over the minimal spanning tree.

In their algorithm for self-stabilizing spanning tree constructions, Huang and Yu, construct a spanning tree, initiated and sourced at a root, and each node containing two variables, L(i) and P(i), denoting the level and the parent of some node i. The level is defined to be the distance of the node from the root node. The range of the level variable is from 0 to N-1, where N-1 is the diameter or maximum-cost path of the graph. The graph is said to be legitimate if all the nodes in the graph establish a parent-child relationship sourced at the root, and for every node other than the root, the level variables is one greater than the level value of their parent. If any of the relationships is not held, then the state is termed to be illegal.

The global legal state (GST) is defined as:

\[ \text{GST} = (\forall i, p: i \neq r \land p = P(i): L(i) = L(p) + 1) \]

The rules to stabilize the system, or the actual moves allowed by the system are defined as:

(R0) \( L(i) \neq n \land L(i) \neq L(p) + 1 \land L(p) \neq n \rightarrow L(i) := L(p) + 1. \)

(R1) \( L(i) \neq n \land L(p) = n \rightarrow L(i) := n. \)

(R2) \( k \in N(i) \land L(i) = n \land L(k) < n-1 \rightarrow L(i) := L(k) + 1; P(i) := k. \)

The definition of the rules and the establishment of the GST can be referenced from the original paper. The algorithm is applicable for a case where the
requisite tree is any self-stabilizing spanning tree. Note that the same result can be achieved by any self-stabilizing BFS algorithm, as proposed by Huang [4] and Srimani [10]. The only difference between the spanning tree and BFS self-stabilizing algorithms specified by Huang is that in the latter case (BFS tree), there is a rule to ensure that a node with multiple children, and of minimum distance away from the root among all the children’s neighbors, remains the parent. In realistic cases the associated cost with edges is not the same, and thus we want to establish a minimal spanning tree (MST) that connects the nodes, for the eventual broadcast of the message from the root.

2.2 Variables

In addition to the variables defined in the original algorithm, L(i) and P(i), we need a parameter that defines the spanning tree to be minimal. This is achieved by checking the node’s local database for the minimal cost edge connecting itself to its parent. We use the notation ij where i is a neighbor of j and thus signifies a path between i and j. Min(Ni) represents the minimum distance cost edge between i and at least one of its neighbors. In cases of a tie, we choose the minimal edge at random.

2.3 Broadcast Algorithm

The broadcast routing utilizes a spanning tree. The construction of the spanning tree which is self-stabilizing is described, and extended to include self-stabilizing minimal spanning tree with broadcast messages arranged in increasing order.
2.3.1 MST Algorithm

We introduce the following rules to make the spanning tree algorithm one that creates a MST:

(R3) \( \text{ip} \neq \text{Min}(N_i) \land \text{p} \neq \text{P(i)} \rightarrow L(i) = L(p) + 1 \)

(R4) \( L(p) = L(i) - 1 \land \text{p} \neq \text{P(i)} \land \text{ip} = \text{Min}(N_p) \rightarrow p = \text{P(i)} \)

R3 establishes from the previous rules a way to identify the minimal edge connecting neighbors \( i, p \) and correcting the level with respect to the distance from the root. Thus any node finding itself to be a child of any node with a higher cost than any edge connecting it to any of its neighbors, changes itself to be a child of the neighbor. R4 lets the node recognize itself as a parent once R3 is committed.

The above two rules alone can lead to a deadlock situation. To break this deadlock situation, a random method can be used to determine which one of the two nodes is to be made parent. The method of to break the deadlock is by choosing a node based on their unique node ids, in this case a node with the greater id involved in a deadlock assigns itself as the parent.

(R5) \( p = \text{P(i)} \land i = \text{P(p)} \land (p > i) \rightarrow i \neq \text{P(p)} \).

Note that node \( i \) is not privileged by either R3 or R4 now, and the node is connected to \( p \) by the minimum cost edge.

2.3.2 Broadcast Hash Algorithm

To incorporate the layer of broadcasting on top of the MST algorithm the following rule is needed:

(R6) \( p = \text{P(i)} \land M.i \neq M.p+1 \lor M.i.source \neq M.p.source \rightarrow M.i := M.p+1, M.i.source := M.p.source. \)
R6 depends on the earlier rules for establishing the correct parent-child relationships between the nodes. However, the privilege or move does not have to wait till the end of the MST algorithm to determine the hash structure of the broadcast messages. R6 guarantees at the end of the stabilization to have a broadcast message reach all the members in the system, and checks this by verifying that a hash structure is maintained.

2.4 Proof of Correction

The proof of the algorithm is established as a whole instead of proving each part of the algorithm. The combined algorithm is as follows:

**Definition 2.1** \( \forall i, p: i \neq r \land p = P(i) \land L(i) = L(p) + 1 \land ip = Min(N_i) \land M_i = M_{p+1} \)

(R0) \( L(i) \neq N \land L(i) = L(p) + 1 \land L(p) \neq N \land ip = Min(N_i) \rightarrow L(i) := L(p) + 1 \)

(R1) \( L(i) \neq N \land L(p) = N) \lor ((i = P(p) \land ip \neq Min(N_p)) \rightarrow L(i) := N \)

(R2) \( L(i) = N \land L(k) < N-1 \land ik = Min(N_i) \rightarrow L(i) := L(p) + 1, P(i) := k, k \in N_i \)

(R3) \( M_i \neq M_p \lor M_i.src \neq M_p.src) \land p = P(i) \rightarrow M_i := M_p \)

**Lemma 2.2** When the system is in the global legal state, deadlock occurs (i.e., no node has a privilege).

**Proof:** This follows from Definition 2.1 and the algorithm. No node has a privilege when the global legal state is satisfied, and thus the system is stable.

**Lemma 2.3** If any node is in an illegal state, at least one privilege exists, and thus the node will make a move.
Proof: By contradiction. Assume that a node does not have a privilege when it is in an illegal state. An illegal state is defined as: for some node i, with node j as its parent, (i) the level of Node i is not one greater than that of its parent, \( L(i) \neq L(j) + 1 \) or (ii) \( L(i) = L(j) + 1 \), but \( i \neq \text{Min}(N_i) \) or (iii) \( M_i \neq M_j \) or (iv) \( L(i) = N \).

For (i) the only reason for node i not having a privilege to move to another state is that no node has chosen it as a parent, a contrary to the assumption that node \( j = P(i) \). If node j was not chosen as a parent, then (R2) has to apply, in which case a privilege exists, a contradiction. For (ii) the parent-child relationship exists, but the edge connecting them may not be one of minimal cost. If a privilege does not exist for node i, the choices available for this to be true are (a) there exist no more neighbors, or (b) the parent-child relationship is wrong. In case (a) \( i \neq \text{Min}(N_i) \) and thus a privilege exists due to R1, in the case of (b), depending on the value of \( L(i) \), R0, R1 or R2 has to be true, both contrary to the assumptions. For (iii), some node(i) existing without a privilege in an illegal state, i.e. without the source id of \( M_i \) being same as its parents value or \( M_i \neq M_P + 1 \), the parent-child relationship must not be set or \( P(i) \) is set but not \( i \) is incorrect. In both these cases, rectifying move is necessitated by rules defined in R0, and R1. Following R0 or R1, R3 has to be true, since the values of \( M_i \) have not been changed by the assumption that (iii) does not draw a move, a contradiction.

Lemma 2.4 The system will resolve all erroneous paths in finite time.

Proof: In the worst case, the nodes all start from an arbitrary value bearing no relationship to the actual level of the nodes in a tree. The concept of a pseudo-move is introduced, defined as those moves which change the state of the system based on the rules, but which may not be a move towards a legal state (for example, R1 in our algorithm is a pseudo-move, designed to create another illegal state, from whence the node can move to start executing actions that lead it to stabilize) The
number of these pseudo-moves is finite, bounded by the diameter of the graph. In the above algorithm, after a sequence of $N-1$ steps, in the worst case all the nodes have reached a state where the values of their levels are all $N$. No steps of the form $R1$ can now be take, the remaining options being $R0$ or $R2$. $R3$ can also lead to pseudo-steps, of the magnitude $M_i+N-1$, i.e. changing their $M_i$ value up to the start message id plus the diameter. After this point the only options are corrective measures, i.e. $R0$, $R2$ or $R3$. The execution of these is from top down, the node with level $N$ nearest to the root responding to its privilege first, followed by those nodes below which change their values with respect to their parents level using steps $R0$ or $R2$. $R3$ can be executed any time, but the convergence to the right value is reached within $N-1$ steps, at which point there are no more privileges. Once this occurs GST is satisfied, and the algorithm is deadlocked.

**Theorem 2.5** This algorithm meets the properties of self-stabilization.

**Proof:** This is proved if properties $P1$ and $P2$ are satisfied. By Lemma 2.2, $P2$ is shown to be true. $P1$ is shown to be true as a consequence of Lemma 2.3, and Lemma 2.4.

### 2.5 Conclusion

This chapter presents a self-stabilizing broadcast algorithm using a minimal spanning tree rooted at some root $r$. The algorithm modifies existing spanning tree algorithm and introduces steps to guarantee the construction of the minimal spanning tree and the verification of message reception at each node. Thus when the system is forced into an illegal state by a transient error, like a node that goes down and comes back up, or some message corruption in a channel, the algorithm automatically starts to stabilize the system to a legal state. The algorithm has several
advantages over traditional broadcast algorithms. Traditional broadcast algorithms, like flooding involve excessive messages being transmitted, are synchronous and susceptible to errors undetected until the next synchronous cycle. In self-stabilizing systems, when any error occurs, the illegal state is identified immediately, and steps taken locally to rectify the erroneous state.
Chapter 3

Multidestinational Routing

Traditional multidestinational routing algorithms assume a packet, a unit of data being transferred from one node to another in a network, containing the information traveling around the network and each node checking the packet header info to see if it is one of the intended destinations. If a node finds that it is one of the destinations, it accepts the information, and retransmits for pick up by the other destinations. More sophisticated algorithms let the source nodes decide which line to retransmit the data packet by determining the link along which the current node believes the next destination is located. In this case, the node generates a new copy of the packet for each of the output lines to be used and includes in each packet only those destinations that are use to use the line. The need to route along the "best" output line to the other destinations assumes the existence of a shortest path tree existing from a source node to one or many of the destination nodes. This research will present an algorithm layered on a self-stabilizing shortest path tree to enable self-stabilizing multidestinational routing in a graph.

3.1 Shortest Path

There are a number of routing algorithms present and being used currently in different network architectures. The simplest and easiest to understand is the shortest path routing. The idea is to build a graph on based on minimizing distance
between a source node and a destination node. The metric used can be a function of
hops (the number of nodes being crossed), time (the transmission delay, queuing
delay), communication cost, or a mix of different metrics determined to give an
optimized measure of network performance between two nodes. Several algorithms
for determining the shortest path between two nodes of a graph are known. The
simplest is an algorithm also proposed by Dijkstra (also called the forward-search
algorithm). The idea is to find the least-cost path from a given source node to all the
other nodes. The algorithm proceeds in stages, with stage \( k \) having determined the
shortest paths to the \( k \) closest nodes from the source. Intermediate steps provide a
way for the system components to change their distance vectors to their source node
if they find a shorter path (or path of least-cost). There are several methods of
creating shortest path trees in a distributed, asynchronous environment. These
include Dijkstra's method, Floyd-Warshall algorithm, and the Bellman-Ford
algorithm[11]. However, the distributed algorithms still assume no transient
failures, and proper initialization prior to the algorithm execution. Huang [7] has
proposed an algorithm for the self-stabilization of shortest path tree constructions.
The self-stabilization application to this type of routing functions in reducing the
extent of initialization, and assumptions about transient failures.

3.2 Variables

Huang and Tsai define three variables needed for the creation of a shortest
path tree rooted at some node \( r \):

\[ p.x : \text{a variable holding the value of } x \text{'s parent, the parent being the next node on the shortest path from } x \text{ to } r. \]

\[ m.x: \text{holds the distance from } x \text{ to } r \text{ (} m.r = 0 \text{)} \]

\[ m.y.x: x \text{'s snapshot of the content of } m.y, \text{ for every } y \in N(x) \]
The algorithm is defined on the basis that the shortest path from x to r is equal to the minimum of each neighbor of x to r summed to the distance between y and x. Each node in the system uses a snapshot taken of its neighbors distance value, this snapshot also has to be stabilized, since any snapshot used out of context can lead to erroneous state (which might be legal). Taking into account both these factors leads to a global legal state defined as follows:

GST: \[ \forall x, y, z : (x \neq r) \land (y, z \in N(x)) \land (m.z.x + d(x,z)) = \text{Min}(m.y.x+d(x,y)) : (m.x=m.z.x+d(x,z)) \land (p.x \in Z) \land (m.y = m.y.x) \]

The allowed privileges are:

(R1) \[ m.x \neq m.z.x + d(x,z) \rightarrow m.x := m.z.x + d(x,z) \]

(R2) \[ (p.x \notin Z) \rightarrow p.x := z; \text{ (z may be any member of Z)} \]

(R3) \[ \exists y : y \in N(x): m.y.x \neq m.y \rightarrow m.y.x := m.y \]

The definition of the rules is easy to derive with respect to the GST, and further justification is provided in the original paper.

### 3.3 Multidestinational Routing Algorithm

For the purpose of multidestinational routing in a graph, another set of rules is necessary to ensure proper delivery to all members of the destination. The following rule specifies that for all the member destinations of the messages, and the nodes intermediary to the destination:


(R5) \[ i = N(p) \land p \neq P(i) \land A[p].i \neq 0 \rightarrow A[p].i = 0 \]

The above two rules generate a list of children that need to accounted for in each parent, and thus provide the parent with a list of destinations available through each
one of its children. The second rule clears up those entries in a parent with illegal values, i.e. an entry for a node which is not a child.

(R6) \[ p = P(i) \land (M_p.\text{dest} \in A[i].k) \land (M_p.\text{dest} \in A[p].i) \rightarrow M_i = M_p \]

R6 verifies that \( i \), some neighbor of \( p \), contains as its descendant, the destination node. Thus R4-R6 works in transmitting the messages to the destination after the guaranteed shortest path tree construction rooted at the source node \( r \) by rules R1-R. R4-R6 fills each parent with a list of descendants, and ensures proper delivery of the data to the intended destinations. The global legal rule is expressed as below for the entire algorithm:

### 3.4 Proof of Correctness

**Definition 3.1** \((m.y = m.y.x) \land A[p].x = A[x] \land (\forall \text{dest } M_{\text{dest}} = M_r)\)

**Lemma 3.2** When the system is in the global legal state, deadlock occurs (i.e., no node has a privilege).

*Proof:* This follows from Definition 3.1 and the algorithm. No node has a privilege when the global legal state is satisfied, and this the system is stable.

**Lemma 3.3** If any node is in a illegal state, at least one privilege exists, and thus the node will make a move.

*Proof:* By contradiction. The message transmission passage and updating of the local routing database is applied on top of a self-stabilized shortest path tree. In this case all the parent-child relationships are assumed to hold as far as the shortest path algorithm is concerned. For the purpose of the multidestinational algorithm which uses the tree, the parent-child relationship exists, and assumed correct (i.e.
there is no additional parent-child correction needed). Given that the illegal states for the multidestinational routing algorithm are: (i) forsome \( A[p] \neq A[i].i \), (ii) \( i = N(p) \land p \neq P(i) \land A[i] \) exists, and (iii) \( p = P(i) \land M_{p,dest} \in A[i].k \land M_{p,dest} \in A[p].i \land M_i \neq M_p \) If (i) is true, and some node \( i \) is not active, this implies that the parent-child relationship has to be wrong. This is contradictory to our assumptions that the underlying shortest path tree exists, and the parent-child relationships well-established. Thus for some node \( i \) with the right level and parent-child relation, \( i \) has to be active, a contradiction. (ii) is a contradiction by the same reasoning. For (iii) to be inactive, and the guard true, the possible scenarios are: a) the parent-child relation is correct, but the values of \( A[i].k \) and \( A[k] \) are both wrong. If this were the case then rule R4 would apply, and a privilege exists. or b) the parent-child relationship is wrong which leads to situation and contradiction depicted in (i) and (ii).

**Lemma 3.4** The system will resolve all erroneous paths in finite time.

**Proof:** Given the shortest path tree the task is to show that the application of R4, R5, and R6 can occur for only a finite number of times. During the multidestinational routing stage of the protocol, there can be no pseudo-moves (as defined in chapter 2) due to precise nature of the parent-child relationship constructed by the shortest path tree algorithm. Thus R4, R5, and R6 are bounded by \( N-1 \) applications. R5 purges those entries that are irrelevant due to their non-applicability after the shortest path tree establishment. R4, R6 are bounded by the level of the created shortest path tree, with the maximum number of steps being \( N-2 \) (the correction steps progress bottom-up, one level above the leaf nodes).

**Theorem 3.5** This algorithm meets the properties of self-stabilization.
Proof: This is proved if properties P1 and P2 are satisfied. By Lemma 3.2, P2 is shown to be true. P1 is shown to be true as a consequence of Lemma 3.3, and Lemma 3.5.

3.5 Conclusion

Chapter 3 introduces an algorithm to route some packet of information to multiple destinations, the routing being self-stabilized. In case a node experiences a transient failures or message corruption, the system reorganizes itself automatically to re-stabilize. In traditional multidestinational routing algorithms the disadvantage is that a token is passed that carries the destination, and left to be picked up by the correct destination(s). The effort to ensure proper delivery is left to underlying layers, or the node themselves recognizing an error through some complex algorithm or wasteful acknowledgments. The advantage of the self-stabilizing routing is that the appropriate node or nodes merely establish themselves in the shortest path tree, and receive the data intended for them from their predecessors. Wasteful transmission is eliminated by the algorithm, in recognizing a global state has been reached when all the destinations receive the data, and no more copying is necessary. It should be noted that in the worst case, when the destination nodes are all leaves of a shortest path trees, the number of transmissions is still less, since only the channels actually identified as leading to the destination are used.
Chapter 4

Connectivity Identification

In any given network, the robustness of routing from one point to another point is dependent on the existence of paths available between them, either directly or through intermediary nodes. The concept of biconnectivity, and strongly connected components define the failures or absence of failures in networks. The detection of biconnected components, and strongly connected components are determined by the creation of depth-first search (DFS) trees in the network. DFS trees have been popular in early research studies, and well-established sequential algorithms exist. The strategy behind a DFS is to traverse the longest or deepest possible path when possible, as compared to a BFS tree which searches the nearest paths first. Most DFS trees include the idea of back-tracking in a network, to establish back-edges and articulation points to create the DFS [13]. This method of creation does not bear well in a distributed environment, where asynchronous events complicate the operation of the algorithm. Most distributed algorithms for the DFS creation restrict themselves to synchronous events [11], and use backtracking as the main method of establishing the search tree. Huang [6] has introduced a self-stabilizing algorithm for circulating a token in a graph in a depth-first fashion. This research introduces a different algorithm to create the DFS tree, using several layers of self-stabilization, including the creation of a directed acyclic graph (DAG).
DAGs are useful in networks and graphs to show precedence among nodes or events. A DAG can be defined in terms of DFS, as a directed graph on which a DFS yields no back edges, where a back edge is an edge connecting some node to one of its ancestors. Ghosh and Karata [9] define a self-stabilization DAG-generation solution, as part of their self-stabilizing algorithm to color planar graphs. The algorithm and proof are defined in the original paper. The algorithm to stabilize a DFS tree incorporates the DAG generation algorithm, to create the tree. This algorithm is in contrast to Huang’s method of creating the DFS tree which involves backtracking, and use of graph coloring methods. The advantage of this research’s method is that there is no back-tracking involved, and the creation of a DAG leads to several unique advantages, and applications.

4.1 Variables

The DAG generation algorithm in Ghosh [9] associates a variable $x[i]$ used to track the direction of each edge from a node. In addition they also define $\text{out}[i]$, the out degree of a node $i$, and set of $x[i]$, representing the set of $x$-values of nodes in $\text{succ}[i]$. $\text{succ}[i]$ is the set of nodes each of which is connected with an outgoing edge from node $i$.

In addition to the above variables required for the DAG generation, we define the following variables and constants needed for the creation of a DFS tree:

- $\text{IN}(i)$ - is the constant set of source nodes of incoming edges to some node $i$
- $L(i)$ - is the level or distance variable of a node $i$ from the root
- $N_i$ - is the constant set of neighbors of $i$
- $ik$ - is the constant directed edge between a node $i$ and its neighbor $k$
- $P(k)$ - identifies the parent of $k$
4.2 DFS Algorithm

**Definition 4.1**

\[ p = P(i) \land IN_i \neq 0 \land L(i) = L(p) + 1; \forall r: IN_r = 0 \land L(r) = 0 \]

Definition 4.1 describes the legal state as a system deadlocked as DFS forests, with roots having no incoming edges, and levels of 0, and for the rest (i.e. not roots) with incoming edges, levels being exactly one greater than that of their parents level variable.

\[ R1 \quad IN_i = 0 \land L(i) \neq 0 \rightarrow L(i) = 0 \]
\[ R2 \quad L(i) \geq N \lor \{ (IN_i \geq 0) \land (L(i) \neq L(p) + 1) \} \rightarrow L(i) = -\infty \]
\[ R3 \quad L(i) \neq L(p)+1 \land p = P(i) \land L(p) < N \rightarrow L(i) = L(p) + 1 \]

R1 functions in assigning those nodes with no incoming edges as roots of their own sub-trees. This allows for the construction of depth-first forests, rooted at several nodes, each of the root by definition containing on incoming arcs. R2 is the error deducing rule, marking as illegal any node that does not register itself as a level greater than that of its parent level or some node whose level is greater than some known limit, for example the diameter of the graph. R3 is the correction rule, setting a nodes level as one greater than that of the nodes parent.

4.3 Proof of Correctness

**Lemma 4.2** When the system is in the global legal state, deadlock occurs (i.e., no node has a privilege).

*Proof:* This follows from Definition 4.1 and the algorithm. No node has a privilege when the global legal state is satisfied, and this the system is stable.

**Lemma 4.3** If any node is in an illegal state, at least one privilege exists, and thus the node will make a move.
Proof: By Contradiction. The possible illegal states are (i) if \( IN_i = 0 \land L(i) \neq 0 \), (ii) \( L(i) \neq L(p) + 1 \land p = P(i) \), and (iii) \( L(i) \geq N \lor \{ (IN_i \geq 0) \land (L(i) \neq L(p) + 1) \} \).

In case (i), if the node has no privilege despite the illegality of the state, the only possibility is that the node information is corrupted. The prevention of corruption of neighbor information contradicts that possibility. In (ii), a node in that illegal state and no privilege has to be a result of incorrect parent-child relationship, again contrary to the presumption regarding the original DAG generation algorithm. In (iii), a node has to execute a privilege if the level of the node is \( N \), since there is not a choice dependent on any neighboring vertex. A node with the second part of (iii) being true and not set for a move assumes incorrect relationship with its parent or incorrect local information regarding its neighbors or incoming edges. The latter is stated to be disallowed as in (i), and the former is a contradiction to the assumption of the DAG generation algorithm.

Lemma 4.4 The system will resolve all erroneous paths in finite time.

Proof: With the creation of the DAG, there can only be a finite number of privileges executed through the application of rules R1, R2, and R3. R1 applies to only those nodes which do not possess any incoming channels. In the simplest case every node can utilize rule R1, thus creating a Depth-First forest of size \( N \). In any of the other scenarios the level of \( L(i) \) cannot change cyclically, since the application of R1 cannot effect or induce the application of R2 and vice-versa, the same holding true for R3. Thus R1 cannot be true for any node after one application, given the DAG structure holds. R2, and R3 are mutually inclusive reciprocally, i.e, the application of one makes impossible the other privilege in the same node. R2 is applied as a pseudo-move which causes a future move through R3. That being, the worst case for the application of R3 is \( N-1 \) times. This stands given that in the
worst case, all nodes have illegal values held in their level variables. R3 cannot re-effect R2 since the application of R3 occurs only if the correct levels and parent-child relationship exists. After the worst case of R2, the only viable alternative as a move is R3. The worst case presents 2N-2 number of moves (including N-1 pseudo-moves). The rationale for this is evident when one considers after R2 has been applied, the maximum number of applications needed to stabilize the system is the number of illegal nodes (N-1, due to R2). Thus the algorithm is guaranteed to stabilize within a finite number of moves of R1, R2, and R3.

**Theorem 4.5** This algorithm meets the properties of self-stabilization.

*Proof:* This is proved if properties P1 and P2 are satisfied. By Lemma 2.2, P2 is shown to be true. P1 is shown to be true as a consequence of Lemma 4.3, and Lemma 4.4.

### 4.4 Conclusion

This chapter describes the algorithm to create depth-forest search forest rooted at some arbitrary node(s). The self-stabilizing nature of the algorithm works in dynamically establishing the forest, and automatically adapting to failures in neighboring links or nodes. The advantage of a self-stabilizing DFS algorithm over traditional algorithms are multi-fold. One main advantage is realized considering that almost all traditional DFS algorithms use some method of backtracking up a tree. This is cumbersome, tedious and inefficient in a distributed, asynchronous system. In traditional systems, each edge has to be encountered twice due to backtracking. In many problems, it is critical that an edge not be traversed twice. Self-stabilizing algorithms do not use backtracking of any sort, and in the algorithm presented in this chapter, the self-stabilizing DAG generation algorithm presents a
layered self-stabilizing approach to creating a DFS forest or tree. In both instances
the decisions to change the system state are made locally, and attributed to some
global illegal state recognized in the system by the individual nodes. This presents a
unique and efficient exercise towards solving the DFS problem.
Chapter 5

Conclusions

Self-stabilization is a developing paradigm in the area of asynchronous, distributed fault-tolerant computing. The idea was proposed by Dijkstra as a strategy to infuse local, and automatic decision making in a distributed system solving the mutual exclusive problem. There have been various extensions to the original paper, in part of the tremendous consequences the application of self-stabilization has to offer in a multitude of areas. Already the theory has been applied to graph-theory, real-time computing, graphics programming, database synchronization, deadlock detection, prevention, and resolution, etc.

This thesis applies self-stabilization, and offers a way to multi-layer algorithms that are auto-stabilizing. The extension of this technique to network algorithms is a natural one, given the dynamic nature of the network processes, and parameters. In particular, the important area of routing is studied including the creation depth-first search forests or trees which are extremely useful in detecting connectivity in a graph, the routing of messages to multiple destinations in a given graph, and the broadcast routing of messages to all nodes in a given graph.

Chapter 2 discusses the traditional broadcast routing, and the difference in approach and effect thereof a self-stabilization algorithm provides as a solution. The algorithm constructs a self-stabilizing minimal spanning tree with a hashed broadcast mechanism. This provides for a logical resolution of message passing,
and verification of message receipt among all the nodes. Only when all the nodes receive the same message does the algorithm reach a deadlock.

Chapter 3 describes an algorithm that passes on information to a subset of the nodes in the graph. The algorithm reaches a deadlock when and only when all the desired destinations receive the same message. The deadlock is a result of the global state description. The layered approach proposed in this chapter provides a way without affecting the nature of self-stabilization, to create multi-protocol self-stabilizing algorithms.

Chapter 4 discusses a method of creating a depth-first tree or forest. The approach once again is to layer the DFS algorithm on a previously generated DAG, also self-stabilizing. The advantage of this approach is the drastic reduction in heavy traffic otherwise realized through traditional algorithms for the creation of DFS, and the unnecessary back-tracking involved.

There are several compelling reasons for applying self-stabilizing algorithms, and heir furthering their study, main among them being:

- The algorithm does not need any initialization
- Any resource or system parameter change automatically creates a privilege in the system. Once the system reaches the global legal state, there is no outstanding privilege.
- The local variables require no initialization.
- The execution of the privileges can be in any order, being true to the distributed, asynchronous nature of a system.
- The algorithm is not susceptible to any transient errors, and recovers from any such situations automatically.
The simple nature of self-stabilization, highlighted by the reasons above makes algorithms easy to implement, and at the same time provide a way for highly fault-tolerant networking system.
Bibliography


