HIGHLY PARALLEL DISTRIBUTED ALGORITHM FOR SHORT­
EST PATH ROUTING IN DYNAMICALLY RECONFIGURABLE
NETWORKS

by

I. Chlamtac, A. Elazar and Y.I. Gold
Technical Report #461
July 1987
Highly Parallel Distributed Algorithm for Shortest Path Routing in Dynamically Reconfigurable Networks

by

I. Chlamtac, A. Elazar and Y.I. Gold

Abstract

In a distributed computer network, nodes participate in the routing process, making routing decisions based on information stored in their routing tables. We present a distributed algorithm for the construction and adaptive maintenance of these routing tables, necessary for shortest (or best weight) path routing in Dynamic Computer Network. In general, these are networks in which changes in topology may occur due to unreliable links, nodes, or due to node mobility, as in the case of mobile radio networks. The presented algorithm is based on the construction and maintenance of a spanning tree, which induces guaranteed shortest path routing tables. The resulting routing adapts to any number of concurrent topological changes in the network within finite time, using proven bounded number of messages and remains loop-free at all times.

The algorithm is unique in that, contrary to existing solutions, it requires that each table update involves the participation of the minimal number of nodes adjacent to the topological change. This optimality property addresses a central issue of reconfigurable networks and leads to time efficiency and highly parallel operation of the algorithm.

* This author was funded in part by grant No. DAAG29-85-K-0044 from the U.S. Army Research Office.
1 INTRODUCTION

The problem of routing over shortest paths is a classic problem in computer networks [1]. In many circumstances it is desirable to provide such routing in networks in which topology changes dynamically. Such changes may result from link or node failures, which occur with nonnegligible probability in a very large network. Topological changes are also inherent to mobile networks, such as tactical radio networks [13].

The reliability and performance of the communication networks are strongly dependent on their ability to cope with these topological changes, so that in finite time after their occurrence, the network will be able to operate normally. Hence a good routing algorithm should be able to adapt quickly and efficiently to topological changes.

The number of algorithms for shortest path routing with centralized or distributed control is abundant [1,12,11]. However, the recovery from topological changes is difficult to solve with both types of routing control. For well-documented reasons we consider the case of distributed routing. In a distributed routing algorithm, each node participates in the routing process making decisions based on network topology information stored in its routing tables. In this case the major problem is the design of a routing algorithm which can adjust the routes to topological changes, without flooding the whole network with control messages for updating the routing tables as needed.

In the New Arpanet algorithm [12], topology information is periodically distributed among all network nodes. In the Fail Safe Distributed Routing Protocol [5], each topology change triggers a two-phase update process which, in general, results in control messages being received by each node in the network. In the Fail-Safe Routing Algorithm for Multihop Packet-Radio Networks, routing tables are periodically exchanged between neighbouring nodes [6].
In this paper we present a distributed routing algorithm which adapts to arbitrary concurrent changes in network topology in the absence of global knowledge of the topology. The presented solution provides local computation of shortest path routes in a dynamic network, based on a distributed computation and maintenance of spanning trees which induce shortest path routing tables. The algorithm guarantees shortest paths, loop-free routes at all times and in addition is unique in several ways which are significant for the dynamic network environment: The algorithm is optimal in minimizing the number of nodes participating in the execution of each routing table update associated with a topological change. The algorithm does not restrict the number of concurrent updates, an important feature for large networks. Lastly, by allowing topological updates to take place concurrently and continuously, the algorithm removes the need for periodic reorganization of the routing tables.

2 The MODEL, PROBLEM statement and EXISTING SOLUTIONS

2.1 The MODEL

The model under investigation is that of a distributed network of \( n \) nodes, with \( n \) distinct identities \( id_0, id_1, \ldots, id_{n-1} \). The network is represented as an undirected graph \( G=(V,E) \), where \( V \) represents the set of network nodes and an edge \( e \) in \( E \) between two nodes in the graph represents a communication link in a point to point network, or in a wireless environment, it signifies that the two corresponding nodes are in line of sight (LOS). We assume that \( E \) can change continuously due to topological changes but network graph remains connected at all times. Communication between nodes is obtained by message transmissions. We assume that messages arrive correctly after a finite, but otherwise unpredictable delay, and are stored in a queue until processed.
2.2 PROBLEM statement

The proposed routing approach is based on shortest path routing tables derived from a spanning tree. A shortest path spanning tree is a tree that covers all network nodes, such that the routes from every node to the tree root are the shortest path between them. Given a shortest path spanning tree rooted at every node in the network, routing tables can be simply obtained: Given a node x, and a shortest path spanning tree with root node y, the routing table of x contains in the entry for destination node y, the link between x and its father in the shortest path spanning tree rooted at y.

The uniqueness of the proposed approach is in finding a distributed organization of topology information stored in the network's nodes so that upon each topology change this topological information can be updated locally by a totally distributed protocol with maximal localization of the update process.

The topology information organization will be derived from the spanning tree, organized in a way which provides distributed computation in a dynamic network. Performing shortest path routing in the network thus reduces the routing problem to the problem of "correct" maintenance or reorganization of the spanning tree upon a topological change.

2.3 EXISTING-SOLUTIONS

A minimum weight spanning tree construction was previously given in [3] using \( O(n \log n + |E|) \) messages and assuming static network topology. In [4] an algorithm for constructing a spanning tree was proposed for a dynamic network. In [4] a single "central" station is responsible for routes' computation and distribution. While the time for the spanning tree reconstruction was shown finite, no analysis of its maximal length was given.
In addition to relying on one station, in the proposed algorithm any topological change may affect the whole network [4].

The solution in [5] constructs spanning tree from each node in order to provide, per each node, the minimal weight paths to any node in the network. However, here too, any topological change requires the propagation of messages over the whole spanning tree of each affected node.

In the next section we formally define the spanning tree structure of the proposed solution, its maintenance and the shortest path routing induced by it.

3. Formal Description

In our approach each node is a root of a spanning tree. Therefore, we shall refer from now to only one such tree, rooted at the 'SINK' node. The algorithm for all remaining spanning trees will be identical.

Let T be a spanning tree of the graph G, with root 'SINK'.

Definition 3.1: Spanning tree T, in G with root node 'SINK', is shortest path spanning tree iff the following holds for every node in \( T \setminus \{ \text{SINK} \} \): if \( x \) in T and its father in T is y then there exists a shortest path between x and SINK that passes through y.

Comment: It is important to notice that the above definition may be satisfied by several spanning trees. All of them can be used for shortest path routing, while possibly each provides alternative shortest path routes.

A shortest path spanning tree, T, with root node SINK, can be generated by the following rules:
a. SINK is in T.
b. A node x is added to T (with father node y) if: 1) y is in T; 2) x is not yet in T; 3) x is an immediate neighbour node of y.

3.1 Construction and Maintenance of Shortest Path Spanning Tree

In this subsection we present the algorithm: its motivation, the messages used by the algorithm, and an optimization technique, incorporated within the algorithm to reduce the number of generated update messages.

In the presented method, routing tables for shortest path routing in reconfigurable network, are based on shortest path spanning trees. The root of a spanning tree is a SINK node. There is one such spanning tree per each node in the network. i.e., each node is a SINK of one spanning tree. Hence in the network, several identical algorithms (each for different SINK node) are simultaneously executed. Each such algorithm, has its own copy of tables and variables. In contrast to other algorithms that maintains spanning trees, for routing purposes, our algorithm for one SINK make use of tables, produced and updated by identical algorithms for different SINKS (for optimization purposes).

In the following discussion, we will present one such algorithm for a given SINK (the other algorithms are identical). The description is complemented with a demonstration of the algorithm on a sample network, which is depicted in Figure 1.
3.1.1 Algorithm Motivation

In shortest path spanning tree based routing, the shortest path from a node to SINK, passes through the link, "preferred link", connecting the node to its father node in the spanning tree, T, rooted at SINK. Once having the shortest paths length (weight) through each neighbour node to SINK, the node can compute its preferred link in T, to SINK.

These shortest paths length (weight), FIRST DISTANCES, are stored in a table: DISTi (first distances from node i to node SINK through each of node i's neighbour nodes). The minimal value in DISTi is referred to as the first distance of node i to SINK.

Similarly, the node i may be selected as a father node in T by some of node i's neighbour nodes. The first distances of all nodes in the sample network, for SINK, are depicted in Figure 1.

When the preferred link of node i to node SINK, in reconfigurable dynamic network is changed, then the first distances of node i's sons, stored in DISTi table will not be correct anymore, since they rely on i's previous first distance to SINK. Therefore, each node maintains as additional table, of SECONDARY DISTANCES, DIST2i. This table specifies for each son node of i, the son's shortest path length to SINK, which does not passes through the son's preferred link (that is, not through node i). The minimal value in DIST2i is referred to as the secondary distance of node i to SINK. The secondary distances of all nodes in the sample network, for SINK, are depicted in Figure 1.

Using the tables DISTi and DIST2i, the node i can compute its preferred link to SINK, provided that these two tables are correctly updated. Keeping the preferred link to SINK, uptodated, routing tables can be easily constructed: for each node j, the preferred link to j in the routing table, is the preferred link to the root of the spanning tree with root node j.
A new preferred link is chosen when 1) the former preferred link has failed, 2) in the update of the first distance table following the receiving of an update message, a link with an improved first distance compared to the first distance through the preferred link was found, 3) the preferred link is not correct anymore, since the node received over this link a JOIN message.

In each case the new preferred link is obtained by ordering node y's neighbour by their distances to SINK: the first distance (neighbours which are not the node sons) or secondary distance (the node sons). The link to the node with minimal distance to SINK, from the above ordering, is chosen as the new preferred link to SINK.

The algorithm is activated as a result of message receiving (message based algorithm). The algorithm updates tables and variables and (possibly) send update messages to other nodes.

3.1.2 The Algorithm Message Types

Four update message types are used within the algorithm. Let the message be received in node x over the link 1, connecting nodes x and y, then:

1. JOIN: node y request to become a son of node x.
2. PATH_LENGTH_LENGTH: node y notifies node x on changes in node y's first and/or secondary distances.
3. NEW_LINK: node x is informed on a new neighbour node, node y. The message contains node y's first and secondary distances to SINK.
4. FAIL_LINK: the Data Link Layer of node x, detects link failure over the link 1.

Each update message contains the first and second distances of the message sending node to SINK and the identity of the node which had detected the topological change (the originating node). These update messages, when received, are used to update the preferred link from the receiving node to SINK.
3.1.3 Algorithm Optimization

The second principle underlying the proposed algorithm is introduced to reduce the number of update messages associated with each change. Notice that in the update process triggered by a topological change, a node say y, may receive several update messages: all of them are routed from the node detecting the topological change (originating node), but received by the considered node over different links belonging to different routes between itself and the originating node.

The principle of the proposed algorithm is to reduce the number of messages sent by y, resulting from update messages received from x. This is obtained by using for the update of a given shortest path spanning tree, the information the node possesses on the remaining spanning trees. Specifically, on receiving an update message from x (x <> SINK) on any link, say l, y computes the new preferred link to SINK, using its preferred link to x on the spanning tree rooted at x.

For this computation y 1) takes the difference between the previous first distance to x over l and the updated one; 2) adding this difference to the first distance to SINK, through the preferred link to x. This results in an updated first distance from y to SINK through a route which passes through node x. If the new preferred link of y to SINK is different from the previous one, then y sends update messages (JOIN over the new preferred link and PATH_LENGTH_UPDATE over the other links). Notice that in this update process, each additional message originated at x and received by y, will result in the computation of the same new preferred link. Therefore, on receiving an additional update messages y will not route additional update messages, to those which resulted from the first message y had received from x.

Given the algorithm motivation and the description of the message types used in the algorithm we proceed with a detailed description of the algorithm accompanied with examples.
3.1.4 Detailed Description Of The Algorithm

The algorithm is message based. Therefore, we outline the algorithm actions taken as a result of receiving each of the above listed message types.

In all the following discussion, we assume that node x receives a message over the link 1, connecting node x to node y.

We demonstrate the course of the algorithm on a sample network given in Figure 1. In this figure, a network graph, a shortest path spanning tree, T, of that network with root node SINK, and the first and second distances of all nodes to the SINK are depicted.

FAIL_LINK : (link 1 has failed)

<1> If 1 = x's preferred link (x has lost its preferred link to SINK) then x:
   1) removes 1 from x's link set.
   2) computes a new preferred link (link with minimal first distance over non son nodes of x or link with minimal secondary distance over son nodes of x (the minimal between the two)).
   3) sends a JOIN message over the preferred link.
   4) if x's first distance to SINK has changed, then x send PATH_LENGTH_UPDATE messages over all x's links but the preferred link. If x's secondary distance has changed, then x send PATH_LENGTH_UPDATE message to its father node over x's preferred link.

<2> if 1 <> preferred link then x:
   5) removes 1 from x's link set.
   6) send PATH_LENGTH_UPDATE messages over x's preferred link if x's secondary distance has changed.

Example:

given the sample network depicted in Figure 1, let us assume that the link connecting nodes b and SINK, link b-SINK, has failed. This results in an update process, which is documented in the following substeps, and depicted in Figure 1a.
STEP 1.1. Following the fail link event, node b loses its preferred link to SINK. Since all the remaining neighbour nodes of b are b's sons then, using the SECOND distance table only, node b chooses node f, b's neighbour node with shortest secondary distance to SINK, as its new father in T. As a result, node b, removes node f from its sons list, sets its preferred link to the link b-f, and sends node f a JOIN message. To its remaining neighbours (node g), node b sends its new distances to SINK (FIRST and SECOND) which are 3 and 4 hops respectively, in a PATH_LENGTH_UPDATE message.

JOIN : (node y wants to become a son node of node x)

<1> If l = preferred link then
   1) x adds node y to x's sons list.
   2) x computes a new preferred link (see actions 2-4 in FAIL_LINK)

<2> If l <> preferred link then
   3) x adds node y to x's sons list.
   4) if x's secondary distance has changed, then x sends PATH_LENGTH_UPDATE message over x's preferred link.

Example :
Continuing the example outlined in FAIL_LINK,

STEP 1.2. The JOIN message from node b is received in f over f's preferred link to SINK. Hence node f's preferred link is not correct anymore. Computing a new one, node f chooses the link f-a as its new preferred link to SINK, sends a JOIN message to node a, and adds node b to f's sons list.

STEP 1.3. Receiving the JOIN message from node f, node a adds node f to a's sons list. Since a's secondary distance to SINK has now changed (from 00 to 6), node a sends a PATH_LENGTH_UPDATE message to its father in T (node SINK).
PATH_LENGTH_UPDATE : (node y updates node x on node y's first and/or secondary distances).

1> If node y's first distance to SINK + 1 < x's first distance to SINK, and l ≠ preferred link then:
   1) if y is a son node of x, then x removes y from x's sons list.
   2) x chooses y as x's new father node (see actions 3, 4 in FAIL_LINK)

2> (x first distance increases)
   If node y's first distance to SINK + 1 < x's first distance to SINK, and l = preferred link then:
   3) x computes its optimal first distance to SINK (see action 2 in FAIL_LINK).
   4) If the result is shorter than x's first distance to SINK, then let z be the new found optimal father node for x, then (if z is a son node of x, then x removes z from x's sons list) x chooses z as its new father node (see actions 3, 4 in FAIL_LINK).
   5) Else x sends PATH_LENGTH_UPDATE message over all its links, but l (which is the preferred link).

3> In all other cases, the update does not change x's preferred link.
   If y is a son of x and now y's first distance to SINK + 1 ≥ x's first distance to SINK, then y is no more x's son:
   6) remove y from x's sons list
   7) send PATH_LENGTH_UPDATE message over x's preferred link if x's secondary distance has changed.

Example:
Continuing the example outlined for the JOIN message:
STEP 1.4. Receiving the PATH_LENGTH_UPDATE message sent by node b (see step 1.1), node g's first distance to SINK changes to 3 hops. As a result, node g sends a PATH_LENGTH_UPDATE message to node d (which contains its new first and second distances to SINK, 3 and 4 hops respectively).
STEP 1.5. Receiving this message node d's secondary distance to SINK is changed. Therefore, node d sends an update message to its father in T, node c. As a result node c's secondary distance to SINK changes too. Hence node c sends an update message to its father in T, node SINK.

Since no new messages are sent then the update process terminates.

NEW_LINK: x adds 1 to its links set.

If y's first distance to SINK + 1 < x's first distance to SINK, then x selects y as its father node (see actions 3.4 in FAIL_LINK).

Example:
Given the network and corresponding first and second distance tables, depicted in Figure 1a, let us assume, that a new link, link a-b, is established between the nodes a and b.

The update process which follows the new link event results in a NEW_LINK message received in the nodes a and b. Node b adjusts its preferred link to the new link, sends a JOIN message to node a and a PATH_LENGTH_UPDATE message to its remaining neighbour nodes (nodes f and g). The point to be made in the following update process is that when node f receives the above message, f notices that node b is no longer f's son (since the first distance of node b to SINK is not greater than node f's first distance to SINK, as it should be for all sons of f in T). Hence f removes node b from its sons list. The network spanning tree, and the corresponding first and second distances tables are depicted in Figure 1b.
3.1.5 Initial Tree Construction

The algorithm constructs a shortest path spanning tree rooted at the SINK. We assume that initially, nodes are aware of their neighbours existence, and their first and secondary distances tables are initialized to infinity. When the algorithm is started, the tree contains the SINK node, which is in distance 0 from itself. SINK initiates PATH_LENGTH_UPDATE messages to all its neighbours, indicating SINK's distance from SINK. On receiving this message nodes do not have a shorter length path to the SINK. Hence, update their first distance table, become the sons of the message sending node (SINK) and inform SINK that it is their father in T, by a JOIN message. This procedure is repeated, at each node: Upon receiving PATH_LENGTH_UPDATE message a node not yet included in the tree updates its first and second distance tables to SINK. The node then chooses a preferred link, as explained earlier. and sends over this preferred link a JOIN message.

It is important to notice that 1) the classic problem in the old ARPANET algorithm, in which overcoming unoptimal paths was slow is avoided, 2) in our approach the update is done from the first and secondary distances using in each case only local data structures.
4. Algorithm Properties

In this section we present the main properties of the presented algorithm. The main results are summarized in two theorems. These theorems indicate that the algorithm is correct and optimal. The algorithm complexity is computed in a third theorem.

To prove the optimality of the proposed algorithm, in the update process following a topological change, we prove that given a topological change in a node x, there is a minimal set of nodes that must be updated. We then prove that 1) the proposed algorithm send update messages to every node in this set 2) no update message is sent to any node not belonging to this set.

The following preliminary definitions and properties are necessary for the theorem proofs.

We view all types of topological change as events of link(s) failure, node(s) failure or new link(s) establishment which result in a non empty set of nodes whose preferred link to SINK or their first distance to SINK has been changed (notice that changes in the preferred link and in the first distance may happen independently). These nodes, AFFECTED nodes of the topological change are defined as follows:

Definition 4.1 : Dependent Set of a node, x, denoted by DS(x) : a node y is in DS(x) iff y is an AFFECTED node of a topological change in x.

We next prove that for a given affected node, node x, only nodes in DS(x) do send messages.
Lemma 1: Given a topological change in a node $x$, a node $y$ sends messages only if $y$ is in $DS(x)$.

Proof: In the algorithm, topological changes are known to a node as a result of receiving a message indicating that. Since it is easy to verify that $y$ is in $DS(x)$ if upon a topological change in $x$, the first or the secondary distance of $y$ to the SINK has changed, we will prove for each message type received by the node $y$, that $y$ send update messages only if $y$'s first or $y$'s secondary distance to SINK has changed. The rest of the proof is therefore given by analyzing the actions taken by the node $y$, upon receiving each of the possible message types, over a link, say 1.

NEW_LINK message: if $y$'s first distance to SINK through 1 is shorter than through the other links of $y$, then $y$ sends a JOIN message over 1, and PATH_UPDATE_MESSAGE to $y$'s other neighbour nodes. Clearly, in that case $y$ belongs to $DS(x)$. $y$'s secondary distance is not affected from this message, since the neighbour of $y$ adjacent to 1, is not $y$'s son. If this message did not shorten $y$'s first distance to SINK, then $y$ was not in $DS(x)$. In the algorithm, this fact is realized by node $y$ in not sending any update messages at all.

FAIL_LINK message: 1) if the failure of 1 does not change either the first or the secondary distances of $y$ to SINK, then $y$ does not send any update messages, and clearly $y$ is not in $DS(x)$. 2) in case 1 was $y$'s preferred link to SINK, $y$ chooses a new preferred link and sends over it a JOIN message. $y$ sends PATH_UPDATE_MESSAGE messages to the rest of its neighbours only if $y$'s first distance through its new preferred link to SINK is different than its first distance to SINK through 1 prior to 1's failure. 3) if prior to 1's failure, 1 connected $y$ to one of its son nodes in $T$, then 1's failure could result in a change in the secondary distance of $y$ to SINK. In that case $y$ sends an update message to its father in $T$. Notice that $y$ is in $DS(x)$, since its first distance through 1 has changed (to infinity). It is obvious that in 2), 3) $y$ is in $DS(x)$.
In Lemma 1 we proved that given a topological change in a node $x$, only nodes within $DS(x)$ do send update messages. We now show that all the affected nodes of the topological change, i.e., all nodes within $DS(x)$, actually receive update messages.

Lemma 2: Given a topological change in a node $x$, every node in $DS(x)$ receives an update message.
Proof: We prove by an induction argument. A node $y$ in distance $k$ from $x$ is either in $DS(x)$ and receives an update message, or $y$ is not in $DS(x)$ and does not receive any update messages. The induction basis is on distance 0, i.e., the node $x$ itself. Since a topological change occurs in $x$, then by the algorithm, $x$ sends update messages, informing its neighbors of its new distance from SINK. If its first or secondary distances to SINK has changed. Let us assume the claim holds for $k \leq n-1$ and prove it for $k=n$. Let us assume that there is a node $y$, in distance $n$ from $x$.

Case 1: If $y$ is in $DS(x)$ then since $y$ is in distance $n$ from $x$, there exists (at least one) a node $z$ s.t. 1) $z$ is a neighbor of $y$, 2) $z$ is in $DS(x)$, 3) $z$ is in distance $n-1$ from $x$. By the induction hypothesis, $z$ received an update message. By Lemma 1 node $z$ send update messages. We must prove that node $y$ actually received one such message. When a node send update messages, there is only one case in which update messages are not sent to all its neighbors, and that is when only its secondary distance has been changed. Hence if only $z'$ secondary distance has changed, and $y$ is not $z$ father then $y$ will not receive an update message from node $z$. If $y$ is not $z'$ father, then let us assume that $z$ is the father of $y$. In that case since only $z'$ secondary distance to SINK has changed, then $y$ is not in $DS(x)$, and contradiction to case 1 assumption. Hence $y$ must be a neighbor node of $z$ which is not $z'$ son or $z'$ father. Again, since $z'$ first distance to SINK was not changed, then $y$ should not be in $DS(x)$, and again a contradiction. Therefore node $z$ send an update message and node $y$ receives this message.
case 2: If \( y \) is not in \( DS(x) \), then 1) assume \( y \) has no neighbour nodes in \( DS(x) \). In that case, \( y \) will receive no update messages since none of \( y \)'s neighbour nodes will send any update messages. 2) Let \( z \) be a neighbour node of \( y \). Let us further assume that \( z \) is in \( DS(x) \). By Lemma 1, \( z \) would have sent an update messages that might have been received by node \( y \). If the update message/messages were not received by node \( y \), then clearly \( y \) will not send any update messages. If one such message was received by node \( y \), then we must prove that either \( y \)'s first distance or \( y \)'s secondary distances to SINK would not have changed. Since \( y \) \( \not\sim x \), then the message types \( y \) can receive are: JOIN and PATH_LENGTH_UPDATE messages. There can be one of two cases: Either \( z \) is a father of \( y \) or not. \( Z \) is in \( DS(x) \), therefore the induction claim holds for \( z \). Hence if \( z \) is \( y \)'s father then \( y \) must be in \( DS(x) \) and contradiction.

If \( z \) is not \( y \)'s father, then the assumption that \( y \) is not in \( DS(x) \) implies that no change in the secondary distance of \( y \)'s sons and the first distance of \( y \)'s other neighbours which belong to \( DS(x) \) would not affect \( y \)'s first or secondary distances. Then even if \( y \) will receive a message from \( z \), then by definition this message cannot change \( y \)'s own distances, hence \( y \) will not produce any update messages.

This two cases complete the induction claim proof.

Q.E.D

Lemma 2a: (Tree property) For a given SINK, the following holds

a) Given pair of nodes: \( x, y \) in \( T \), s.t. \( y \) is \( x \)'s son, then \( \text{DIST}_y = \text{DIST}_x + 1 \) (assuming \( \text{DIST}_{\text{sink}} = -1 \)).

b) In a network with no topological changes and no update messages flowing over the network links, then for every node, say \( x \), \( \text{DIST}_x \) is node \( x \)'s shortest distance to SINK.
Proof:
a) We will use an induction argument on the distance from SINK. For zero distance, \(x = \text{SINK}\) and \(y\) is an immediate son node of SINK. Clearly a) holds, since \(y\) becomes a son of SINK, using a JOIN message, and then (see procedure elect_father) \(\text{DIST}_y = \text{DIST}_x + 1 = -1 + 1 = 0\). Let us assume the induction correctness for nodes up to and including nodes in distance \(K\) from SINK. We will now prove the claim correctness for nodes in distance \(K+1\) from SINK. Let \(x,y\) be nodes in \(T\), s.t. \(x\) is in distance \(k\) from SINK and \(y\) is a son of \(x\). \(\text{DIST}_y \leq K\) cannot hold, since then the induction would hold for \(y\) and then \(\text{DIST}_y > \text{DIST}_x \Rightarrow \text{DIST}_y > K\) and contradiction. Therefore, \(\text{DIST}_y > K\). By the algorithm, a node sends a JOIN message only to its neighbour node. \(x\) is a neighbour of \(y\), hence \(\text{DIST}_y = K+1\).
b) In a static network with no update messages over the network's links, the algorithm is not executed. Therefore, \(\text{DIST}_i\) is fixed in each node \(i\). Let us begin with the node SINK and continue to its descendents. By point a) of the Lemma, \(T\) is a tree. Since no changes occur, then when going down the tree from a node \(x\) to its son \(y\), \(\text{DIST}_y = \text{DIST}_x + 1 = \text{distance of } x \text{ from SINK} + 1\).

Lemma 3 : At the end of the update process triggered by a topological change in a node \(x\), each node in the network has the shortest path to SINK.
Proof : By Lemma 1 nodes send messages only if they are in \(\text{DS}(x)\) and they update correctly their preferred link. In Lemma 2, we proved that by our algorithm all nodes in \(\text{DS}(x)\) receive messages. In Lemma 2a, we proved that the algorithm maintains trees in each network node.

Q.E.D
We next prove a bound on the number of messages transmitted by each node as a result of one topological change. The analysis of the algorithm message complexity is based on this bound.

Lemma 4: Given a topological change in a node x, a node y in DS(x) does not send more than o(Rank(y)) messages.

Proof: Let y be any node other than x in DS(x), and let dy denote the distance of y from SINK. By lemma 2, y will receive an update message (since y is in DS(x)). Let us denote the messages y receives during the update process by \{Mi\}. Clearly, none of these messages will be FAIL_LINK or NEW_LINK, since these message types may occur only in x. Let us analyze the the other message types.

JOIN: A JOIN message on link other than the preferred link will not change dy, but may alter y's secondary distance. Therefore, each such JOIN message can result with one message being send over y's preferred link. Since the number of sons y may have is bounded by Rank(y) - 1, then the Lemma bound holds.

JOIN: A JOIN message over the preferred link results in a search for the minimal first distance to SINK through non son's neighbours nodes or secondary distance through the son nodes. The search is guaranteed to succeed since the network is assumed to be connected at all times. When a new father node for y is elected, update messages are sent to the new father and to the other nodes. Again no more than Rank(y) messages. No more than one JOIN message over the preferred link may occur, hence this type of message contribute no more than Rank(y) messages.
PATH_LENGTH_UPDATE: When this message is received, the algorithm uses the spanning tree rooted by x to determine the real effect of the topological change in x over the preferred link of y to SINK. Since all update message contain x identification (the originating node), then all PATH_LENGTH_UPDATE message received by y, and originated from x, will result in the same computed first distance to SINK, through the route which passes through the node x. Therefore, update message, from y, if any, will be generated as a result of only one received message of the PATH_LENGTH_UPDATE messages in \( \{M_i\} \) (actually the first one). Hence no more than \( \text{Rank}(y) \) messages will be send by y.

Summing the messages sent by y over all the message types listed above, yield the claimed bound correctness.

Q.E.D

Theorem 1, below, summarises the correctness and the main properties of the algorithm.

Theorem 1: Given a topological change in a node x, all nodes in DS(x) receives an update message. No more than one update message is sent by a node in DS(x) over each link. Moreover, T induces shortest paths routes from each node to the SINK.

Proof: By Lemma 1,2,2a,3,4.

Q.E.D

We next compute the complexity of the algorithm, as a function of the maximal number of update messages generated in an update process triggered by a topological change.
The previous results proved the correctness of the algorithm in face of one topological change. We now show its correctness for any combination of concurrent topological changes.

Theorem 2: The message complexity of the update process resulting from algorithm activation, following a topological change in a node \( x \) is bounded by:

\[
|DS(x)| \cdot o(\max \text{Rank}(i))
\]

where \( i \) ranges over nodes in \( DS(x) \).

Proof: Every node in \( DS(x) \) sends messages as a result of topological change in \( x \) (Lemma 2). By Lemma 4 no more than \( o(\text{Rank}(i)) \) messages are sent by a node \( i \). Therefore at most \( o(\max \text{Rank}(i)) \) messages are sent by each node in \( DS(x) \), and there are \( |DS(x)| \) nodes in \( DS(x) \).

Q.E.D

Comment: The message complexity of the whole tree construction, is bounded by \( o(N^2) \) (by Theorem 2). This complexity was shown to be the lower bound for any algorithm that constructs a spanning tree [9].

The previous results proved the correctness of the algorithm in face of one topological change. We now show its correctness for any combination of concurrent topological changes.

Theorem 3: Given any combination of topological changes in any order, in the network, the algorithm updates \( T \) correctly, and at the end of the update process, each node has the preferred link to SINK.

Proof: Every node in \( DS(x) \) sends at most \( O(\text{Rank}(x)) \) messages as a result of a topological change in \( x \) (Lemma 2 and 4). By the assumption, messages sent by a node \( i \) are received in a node \( j \) in the same order of their transmission. Therefore, update messages from \( x \) on any later topological changes will be received later at nodes which received the previous messages from \( x \).
Since 1) knowledge of the path length to SINK through each link is sufficient to compute the shortest path to SINK, 2) new messages from the same source node are not received before previous messages. Therefore, in a given moment, the update a node performs is in accordance with the latest message that was sent to it and received by it. Hence, by the end of the update process, each node will have the exact distance of itself to SINK through each of its links. Which results in correct computation of the shortest path to SINK.

Q.E.D

Conclusion : The algorithm uses the minimal number of nodes in the update process, which follows a topological change.  

Proof : Any distributed algorithm, that maintains shortest path spanning trees, must send messages, when a topological change occurs in $x$, to every node in $DS(x)$. We proved in Lemma 1,2 that our algorithm does send messages to every node in $DS(x)$. Furthermore, in Lemma 2 we proved that no messages are received by nodes in $G - DS(x)$. Hence the conclusion correctness.

Q.E.D

Comment : The assumption, that messages sent by a node $i$ are received by a node $j$, in the same order of their transmission, is a convenient assumption for the algorithms correctness proofs. But, in general, this assumption does not hold. However, it is obvious that this assumption can be eliminated, with a minor change in the algorithm (see also [11]). Adding to each node a counter, that the node increments each time it send a message, and storing this number in every message the node send, every node which receives messages from the above node, can differentiate between old and new messages.
4.1 Simulation Results

The considerable complexity of the algorithm and of its correctness proofs have lead us to test the algorithm using a simulation program. The simulation program receives a network graph and the identity of the SINK node. Furthermore, topological events, of the cases considered, (fail link and new link event) can be generated any time during the simulation. The program routes the messages send by nodes to the requested neighbour nodes. Moreover, a trace of the messages flow in the network is depicted. Using this program, the algorithm properties, as given in this section, and the algorithm message complexity were verified experimentally.

5. Discussion

We have presented a distributed algorithm which constructs and maintains shortest paths routes, and is unique in using minimal number of nodes in the update process and a bounded polynomial number of update messages. The algorithm is totally distributed and is based on the construction and maintenance of shortest path spanning tree. The algorithm is applicable for "fail safe" routing in point to point networks and to mobile networks, since it broadly recognizes a topological changes as an event of combinations of link failures and additions.

The discussion has concentrated on topological changes, hence equal link weights were assumed. However, it is easy to see that a minimal weight path spanning tree based on link weights can be derived in a straightforward way, by generalizing a topological change to include a change in the link weight. In this way we obtain distributed minimal weight spanning tree, which is unique in that the update process is executed using minimal number of nodes and does not rely on a periodic update process.
References

7. M.J. Johnson, "Analysis of Routing Table Update Activity after Resource Failure in a Distributed Computer Network", ACM SIGCOMM'83 symposium, Communications Architectures & Protocols, pp 14-20
Appendix: Algorithm Description

Control Messages

The algorithm uses several control messages. We describe each control message type, assuming it has been sent by the node idj and received by the node i. l signifies the link over which node i has received that message (the link connecting nodes idj and i). We assume that the order of messages arrival corresponds to their transmission order. I.e., messages routed by node x to node y will be received in y at the same order they were sent by x.

1. Join (l,idj)
   node idj wishes to become the son of node i.

2. Update_path_length (l,idj,distj,dist2j,sourcej)
   node idj informs node i on its current first and secondary distances to SINK. sourcej indicates which node has originated the update message as a result of the topological change. Since this field is generally not changed by nodes, we omit it whenever it is not changed or inspected by nodes in order to clarify the code.

3. New_link (l,idj,distj,dist2j)
   node i is informed on a new link, connecting nodes idj to itself. For presentation simplicity we assume that the message already includes the information on the first and second distances to SINK through the new neighbour node.

4. Fail_link (l)
   node i is informed by the Data Link Layer that the link l has failed.
For defining the shortest path routing based on T, we introduce the following notations.

For each node:

- **Li** - list of i's neighbours in G
- **Di** - the cardinality of Li
- **SONi** - list of i's sons in the spanning tree
- **FATHERi** - i's father in the spanning tree
- **LINK_TO_FATHERi** - the link through which the shortest path passes from node i to SINK, also called PREFERRED LINK
- **Vi** - vector of distances to SINK of all the neighbours of the node i.
- **DISTi** - the shortest path length from node i to SINK
- **V2i** - vector of secondary distances to SINK of all the neighbours of the node i (relevant only for sons of i in the spanning tree)
- **DIST2i** - the secondary shortest path length from node i to SINK
- **LINK_TO_SECONDi** - the link through the secondary shortest path passes

**PSEUDO CODE**

procedure UPDATE_DIST2(node : node_type);
begin
  OLD DIST2node <-- DIST2node { Save current DIST2 of node}
  DIST2node <-- min V2node (k)
  k in Li-[LINK_TO_FATHERnode]
  if OLD DIST2node <> DIST2node then
    send (LINK_TO_FATHERnode, PATH_LENGTH_UPDATE (IDnode, DISTnode, DIST2node, IDnode))
end { UPDATE_DIST2 }

procedure ELECT_FATHER (l : links; SOURCE : node_type);
begin
  send (l, JOIN(IDi))
  OLDDISTi <-- DISTi
  DISTi <-- V2i(l) + 1
  LINK_TO_FATHERi <-- l { Set preferred link to l }
  IFfatheri <-- ID(l)
  UPDATE DIST2(i)
  if OLDDISTi <> DISTi then
    forall j in Li-[l] do
      send (j, PATH_LENGTH_UPDATE (IDi, DISTi, DIST2i, SOURCE))
end { ELECT_FATHER }
begin { of ALGORITHM }

Vi <-- [ ]
V2i <-- [ ]
DISTi <-- 00
DIST2i <-- 00

Doforever

wait message;
case message of
NEW_LINK(1, distj, distj2, id):
begin { New link between nodes i and id } 
IDi(1) <-- id
Li <-- Li + [1] { Add new link to link set }
Vi(l) <-- distj
V2i(l) <-- distj2
if Vi(l) + 1 < DISTi { i is in distance Vi(l)+1 from SINK through 1 }
then ELECT_FATHER (1, IDi)
end;

FAIL_LINK (1) : { The link 1 failed }
if l=link_to_father then { i has lost its preferred link }
begin
Vi(l) <-- 00
V2i(l) <-- 00
Li <-- Li - [1] { remove 1 from link set }
j <-- Min ( Min Vi(k), Min V2i(k2) )
     k in Li-1 [1] k2 in SONi
ELECT_FATHER (j, IDi)
end
else { other link }
begin { The failed link is not the preferred link }
Li <-- Li - [1]
Vi(l) <-- 00
V2i(l) <-- 00
if i in SONi then SONi <-- SONi - [1]
UPDATE_DIST2 (1)
end

PATH_LENGTH_UPDATE (idj, distj, distj2, sourcej) :
begin { Compute the best link over which this message could have came }

Best_link <-- link_to_father(sourcej)
Vi(Best_link) <-- Vi(Best_link) - (Vi(l) - distj)
V2i(l) <-- distj

if (distj + 1 < DISTi) and (l <> link_to_father) then
begin { i has detect a better link }
if l in SONi then SONi <-- SONi - [1]
ELECT_FATHER (l, sourcej)
end
else if (distj + 1 > DISTi) and (l=link_to_father) then
begin { The path through the preferred link is longer }
DISTi <-- distj + 1
k <-- Min ( Min Vi(r), Min V2i(r2) )
r in Li-SONi-[1] r2 in SONi
if Vi(k) + 1 < DISTi then
begin
if k in SONi then SONi <-- SONi - [k]
ELECT_FATHER (k, sourcej)
end
else { Sending update messages. The preferred link does not change, but the first distance is longer. }
for all k in Li-1 [1] do
send(k, PATH_LENGTH_UPDATE (IDi, DISTi, DIST2i, sourcej))
end
else
begin { A path length update which does not change the preferred link }
   if l in SONi then
      if distj + 1 = DISTi then
         SONi <-- SONi - [l] { The son has found a better link to SINK, which does not pass through node i }
      end
   end
   UPDATE_DIST2(i)
end
end { case }
end doforever
end; { ALGORITHM }
Figure 1: Sample network configuration. The spanning tree edges lines are full, while other edges are in dashed lines. The relevant data structures of each node are enclosed within the table. Notice that DIST2 is shown only for nodes which have sons in the spanning tree.

<table>
<thead>
<tr>
<th>node</th>
<th>dist1</th>
<th>dist2</th>
<th>preferred link</th>
<th>secondary link</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINK</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>A-SINK</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
<td>B-SINK</td>
<td>B-F</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>4</td>
<td>C-SINK</td>
<td>C-D</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td></td>
<td>D-C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
<td>E-A</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td></td>
<td>F-B</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td></td>
<td>G-B</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1a: The sample network following the update process triggered by the failure of the link SINK-B.

<table>
<thead>
<tr>
<th>node</th>
<th>dist1</th>
<th>dist2</th>
<th>preferred link</th>
<th>secondary link</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINK</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>6</td>
<td>A-SINK</td>
<td>A-F</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td></td>
<td>B-F</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>6</td>
<td>C-SINK</td>
<td>C-D</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>5</td>
<td>D-C</td>
<td>D-G</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
<td>E-A</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>5</td>
<td>F-A</td>
<td>F-B</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td></td>
<td>G-D</td>
<td></td>
</tr>
</tbody>
</table>
Figure 1b: The sample network following the update process triggered by the link A-B establishment.

<table>
<thead>
<tr>
<th>node</th>
<th>dist1</th>
<th>dist2</th>
<th>preferred link</th>
<th>secondary link</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINK</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>5</td>
<td>A-SINK</td>
<td>A-B</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td></td>
<td>B-A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>5</td>
<td>C-SINK</td>
<td>C-D</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>4</td>
<td>D-C</td>
<td>D-G</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
<td>E-A</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td></td>
<td>F-A</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td></td>
<td>G-D</td>
<td></td>
</tr>
</tbody>
</table>