March 1994

UILU-ENG-94-2210
ACT-131

Applied Computation Theory

A FAULT TOLERANT DISTRIBUTED ALGORITHM FOR MINIMUM-WEIGHT SPANNING TREES

Reuben Pasquini and Michael C. Loui

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

UNCLASSIFIED.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-94-2210 (ACT-131)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab.
University of Illinois

6b. OFFICE SYMBOL *(If applicable)*
N/A

7a. NAME OF MONITORING ORGANIZATION
National Science Foundation

6c. ADDRESS *(City, State, and ZIP Code)*
1308 West Main Street
Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*
Washington, DC 20050

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
National Science Foundation

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
Washington, DC 20050

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A FAULT TOLERANT DISTRIBUTED ALGORITHM FOR MINIMUM-WEIGHT SPANNING TREES

12. PERSONAL AUTHOR(S)
Reuben Pasquini, Michael C. Loui

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1993 March 11

15. PAGE COUNT
31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
distributed algorithm, fault tolerance,
minimum weight spanning tree

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

We present a fault tolerant algorithm for construction of a minimum spanning tree (MST) over the reliable links of an asynchronous network with n processor nodes and m fail stop links. Up to f of the links may fail before our algorithm begins execution. Nodes communicate by exchanging messages. Our algorithm executes in two phases: Tree Construction phase and a Tree Update phase. The Tree Construction phase of the algorithm is a variation on the MST algorithm of Gallager, Humblet, and Spira, which constructs an MST on the set of links known to be reliable (not faulty) by our algorithm. The Tree Update phase executes an MST update algorithm to include in the MST links that are initially assumed to be faulty but are later found to be reliable. Our algorithm has $O(fn^2)$ message and time complexities.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE *(Include Area Code)*

22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

A Fault Tolerant Distributed Algorithm for Minimum-Weight Spanning Trees

Reuben Pasquini Michael C. Loui

March 11, 1994

Abstract

We present a fault tolerant algorithm for construction of a minimum spanning tree (MST) over the reliable links of an asynchronous network with n processor nodes and m fail stop links. Up to f of the links may fail before our algorithm begins execution. Nodes communicate by exchanging messages. Our algorithm executes in two phases: Tree Construction phase and a Tree Update phase. The Tree Construction phase of the algorithm is a variation on the MST algorithm of Gallager, Humblet, and Spira, which constructs an MST on the set of links known to be reliable (not faulty) by our algorithm. The Tree Update phase executes an MST update algorithm to include in the MST links that are initially assumed to be faulty but are later found to be reliable. Our algorithm has $O(fn^2)$ message and time complexities.

key words: distributed algorithm, fault tolerance, minimum weight spanning tree

1 Introduction

Minimum spanning tree (MST) construction is a fundamental problem for distributed systems connected by weighted links. A minimum spanning tree may be used when one node wishes to broadcast information to all the other nodes of the network. MST algorithms can also serve as building blocks for other distributed algorithms like network synchronization, breadth-first-search, and deadlock resolution [5]. If two distributed problems $P1$ and $P2$ are *equivalent*, then an efficient algorithm to solve $P1$, with a number of messages $M1$, implies an efficient algorithm to solve $P2$, with a number of messages $M2 = O(M1)$, and vice versa [6]. Awerbuch proved that the MST problem is equivalent to a large class of problems including leader election, spanning tree construction, and counting the number of nodes in a network [2].

The majority of distributed algorithms for the construction of MST's on computer networks follow the Borůvka algorithm. A *fragment* X is a subtree of an MST. A link (x, y) is an *outgoing link* of a fragment X if one endpoint x is in X and the other endpoint y is not in X. Initially each node x is treated as a fragment. Each fragment X has a unique minimum outgoing link (x, y). Fragment X merges with fragment Y, the fragment at the end of (x, y), to form a new fragment Z. The Borůvaka algorithm iterates this procedure on the set of fragments in the network until only one fragment exists on the network (i.e., every node belongs to the same fragment Z and Z defines the network MST). Naive distributed implementations of Borůvka's algorithms introduced by Chang, Parker, and Samadi, and by Lavallee and Roucairol have $O(n^2)$ message complexity [6]. Gallager, Humblet, and Spira introduced the first MST algorithm to achieve $O(e + n \log n)$ message complexity (we call this algorithm the GHS algorithm) [7]. Gafni [6], Chin and Ting [5], Awerbuch [2], and Garay, Kutten, and Peleg (GKP) [8] each modified the GHS algorithm to improve its time complexity. The Awerbuch and GKP algorithms attain linear time complexity by executing GHS in two phases.

Tsin [16] took an orthogonal approach to the MST problem, constructing an MST by repeatedly applying a cycle resolution algorithm to a network. This algorithm adds a link to the current tree, removes the heaviest link from a resulting cycle, and iterates over all the links in the network. If all the links are added to the tree and all the cycles are resolved, then the MST results.

The development of fault tolerant algorithms for distributed networks is an active area of research. Several papers address fault tolerant leader election and fault tolerant spanning tree construction [1, 4, 9, 10, 13]. To tolerate faulty links, these algorithms use more messages than non-fault tolerant algorithms to ensure that messages are received by some reliable nodes over some reliable links [1, 9, 13].

In this paper we develop a fault tolerant MST algorithm. We do not know of any previous fault tolerant MST algorithm. Our algorithm incorporates ideas from conventional MST algorithms, tree update after topology change algorithms, and fault tolerant leader election and spanning tree algorithms.

We assume a distributed network of n reliable processor nodes with m bidirectional fail stop links where up to f links may fail before execution of our algorithm begins. Nodes can communicate only via asynchronous message passing. Each node x initially knows the weight of each link (x, y) incident to x, but x cannot determine whether a link (x, y) is faulty

or reliable unless x receives a message over (x, y), or x receives a message stating (x, y) is reliable over a different link (x, z).

Our algorithm modifies the GHS algorithm to run in two phases (fragment merging and cycle resolution), like the Awerbuch and GKP algorithms. We further modify the GHS algorithm by increasing the number of messages issued during fragment merging, also for fault tolerance. While increasing messages alone is sufficient to develop fault tolerant spanning tree and leader election algorithms [1, 9, 13], our MST algorithm requires the cycle resolution stage too; this gives our algorithm a high message complexity. Our algorithm has $O(fn \log n + fn^2)$ message and time complexities.

2 Definitions and Model

We assume each node x has a unique identity $ID(x)$ and that each node knows the identity of its neighbor nodes. We also assume that each link (x, y) is assigned a unique *weight* $W(x, y)$ before execution of our algorithm begins. The weight of a link is a parameter associated with some cost (dollar cost, time delay, reliability) of sending a message over the link. If link (x, y) has a smaller weight than (x, z), then (x, y) is *lighter* than (x, z), and (x, z) is *heavier* than (x, y). Because the weight $W(x, y)$ of each link (x, y) is unique, $W(x, y)$ may be used as an identifier of link (x, y).

If each link does not have a unique weight, then a unique MST may not exist [6]. A network without uniquely weighted links may be handled by this algorithm by making the weight of each link (x, y) a unique ordered pair: $(W(x, y), (ID(x), ID(y)))$.

An *initiator* is a node which spontaneously begins executing our algorithm. A *fragment* is a subtree of the MST. Initially, each node constitutes one fragment. A *fail stop link* is a link that fails by not delivering any messages. If a link delivers all messages in FIFO order without errors, then the link is *reliable*. An *internal link* is a link joining two nodes that are members of the same fragment. An *outgoing link* is a link joining two nodes belonging to different fragments. The *minimum outgoing link of a node* x is the lightest outgoing link incident to x. The *minimum outgoing link of a fragment* X is the lightest link that is a minimum outgoing link of a node in X. We say a node x *broadcasts* a message M if x sends a copy of M to each of the children of x who in turn send a copy of M (perhaps with changes in the parameters carried by M) to each of their children and so on.

We assume an asynchronous network with up to f fail stop links that fail before our algorithm begins execution, not during execution. Nodes communicate by message passing. Because the network is asynchronous, it is impossible for a node x to determine whether an incident link (x, y) is faulty or slow unless x receives a message over (x, y), or some other node z tells x that (x, y) is reliable. In our algorithm, x learns that (x, y) is reliable only by receiving a message over (x, y).

3 Data Structure

Each node x maintains a data structure which stores three types of information: link information, update information, and fragment information.

3.1 Link Information

The weight and the type of each link (x, y) incident on x constitute the link information. There are four basic link types:

- BRANCH
- REJECT
- ASSUMED FAULTY (AF)
- ERROR.

If x believes that (x, y) is an outgoing link from X (the fragment to which x belongs), then x classifies (x, y) as a BRANCH link. When x begins our algorithm, x classifies every link incident to x as a BRANCH link.

If x believes (x, y) is an internal link, then x classifies (x, y) as a REJECT link.

If x believes (x, y) is a failed link, then x classifies (x, y) as an ASSUMED FAULTY (AF) link.

If x classifies (x, y) as an AF link and x learns later that (x, y) is actually reliable, then x classifies (x, y) as an ERROR link.

3.2 Update Information

Our algorithm executes in two disjoint phases: a Tree Construction phase and a Tree Update phase. During the Tree Update phase, each node x initiates a cycle resolution algorithm (CRA) on each link (x, y) that x initially assumes is faulty but x later learns is actually reliable (i.e., the internal ERROR links incident to x). The CRA may execute on only one link at a time. To prevent different nodes from initiating the CRA on different links at the same time, our algorithm uses Raymond's mutual exclusion algorithm [15]. Raymond's algorithm passes a token between nodes. When a node x possesses the token, then x has the right to initiate the CRA.

To execute Raymond's algorithm in a fragment X, each node x in X keeps *TOKEN_HOLDER*, *REQUEST_Q*, and *ASKED* variables. *TOKEN_HOLDER* holds the name of the neighbor of x that is the first node on the path from x to the node currently holding the token. The *REQUEST_Q* is a FIFO queue that records the nodes that request the token from x. Finally, if x has requested the token but x has not yet received the token, then x sets the *ASKED* flag to *TRUE*.

3.3 Fragment Information

Each node x maintains four variables defining the state of the fragment X to which x belongs: $FID(X)$, $LEVEL(X)$, $CHILDREN(x)$, and $PARENT(x)$.

The variable $FID(X)$ is the fragment identity variable; $FID(X)$ uniquely identifies the fragment X to which x belongs. $FID(X)$ contains the $ID(r)$ of the node r which is the root of the tree X defines. Each fragment has a different root; therefore each fragment X has a unique $FID(X)$.

The variable $LEVEL(X)$ records the level of X. The level of X is an integer parameter less than or equal to $\log_2 n(X)$ where $n(X)$ is the number of nodes in X. The level of a fragment is determined during the execution of the algorithm.

If the path from node x to the root r of X begins with node y, then y is the parent of x. The variable $PARENT(x)$ records the the parent y of x. The root r sets $PARENT(r) = root$.

The variable $CHILDREN(x)$ records the nodes adjacent to x in X that are not the parent of x. We refer to the links joining x with its children and parent as *MST links* or *fragment links*.

4 Algorithm

We present a fault tolerant algorithm for construction of a minimum spanning tree over the reliable links of a message passing asynchronous network with n nodes and m links. Up to f of the links may be faulty. Our algorithm tolerates links which fail by not allowing messages to pass over them (fail stop links). A node x can not determine whether a link (x, y) is slow or faulty unless x receives a message over (x, y). Every link failure must occur before the algorithm begins execution. The failure of some links after the algorithm begins execution may cause the algorithm to fail to construct a minimum spanning tree.

If the links connecting two subgraphs of the network fail, then the network becomes disconnected. In this case our algorithm constructs a minimum spanning tree over the reliable links of each subgraph which contains an initiator of the algorithm. We assume through the rest of this paper that the network remains connected so that our algorithm constructs an MST over the reliable links of the whole network.

Our algorithm executes in two phases: a Tree Construction phase and a Tree Update phase. The Tree Construction phase of the algorithm is a variation on the MST algorithm of Gallager, Humblet, and Spira (GHS) [7], which constructs an MST on the set of links known to be reliable (not faulty) by our algorithm.

The Tree Update phase of the algorithm begins execution after the completion of the Tree Construction phase. During the Tree Update phase, our algorithm modifies the MST so that the MST is valid over links that are initially assumed to be faulty but are later found to be reliable.

4.1 Messages

Our algorithm uses several different messages to construct the MST. We present here a summary of each message, its format, and its function. To simplify the presentation of the message formats, we use the following abbreviations:

- $FID(X)$ - This expression denotes the fragment identifier of fragment X.
- $L(X)$ - This expression denotes the level of fragment X.
- (x, y) - this expression denotes the the link joining nodes x and y. When (x, y) is included in a message, (x, y) may be replaced with the weight of (x, y) since each link has a unique weight.

Our algorithm uses the following messages:

- TEST($FID(X), L(X)$)

  A node x in fragment X sends a TEST message over (x, y) to determine the link type of (x, y). When y receives a TEST message from x, node y may respond with a REJECT, ACCEPT, or WAIT message (see section 4.3).

- REJECT

  If y receives a TEST message on link (x, y) from a node x in the same fragment as y, then y responds to the TEST message with a REJECT message. If a node y sends or receives a REJECT message over a link (x, y), then y classifies (x, y) as a REJECT link (see section 4.3).

- ACCEPT($FID(Y), FID(X)$)

  If node y in fragment Y receives a TEST message on link (x, y) from node x in a different fragment X and $L(X) \leq L(Y)$, then y responds to the TEST message with an ACCEPT message (see section 4.3).

- WAIT

  If node y in fragment Y receives a TEST message on link (x, y) from node x in a different fragment X and $L(X) > L(Y)$, then y responds to the TEST message with a WAIT message (see section 4.3).

- ERR

  If node x classifies (x, y) as an ERROR link, then x sends and ERR message over (x, y) to tell y to classify (x, y) as an ERROR link too (see section 4.3.1).

- REPORT((y, z))

  If x believes (y, z) is the minimum outgoing link from x and the descendants of x in X, then x will cite (y, z) in a REPORT message to the parent of x during a search for the minimum outgoing link from fragment X (see sections 4.3 and 4.4).

- CHANGE_ROOT((x, y))

  If (x, y) is the minimum outgoing link from fragment X, then the root r of X causes X to merge with fragment Y over link (x, y) by broadcasting a CHANGE_ROOT message through X. The CHANGE_ROOT message tells the nodes in fragment X to rearrange their parent to child relationships so that the root of fragment Y becomes the new root of fragment X (see section 4.4).

- CONNECT($FID(X), L(X)$)

  If x in fragment X receives a CHANGE_ROOT((x, y)) message, then x makes its new parent y, and x sends a CONNECT message to y. The CONNECT message lets y know that fragment X has merged with fragment Y along link (x, y) (see sections 4.4 and 4.4.1).

- INITIATE($FID(Z), L(Z)$)

  When two fragments X and Y merge to form a new fragment Z, the root r of Z broadcasts an INITIATE message so that every node in Z has correct values of $L(Z)$ and $FID(Z)$ (see sections 4.2 and 4.4.1).

- CORRECT($FID(Y), L(Y), STATE(Y)$)

  If fragment X becomes part of an already existing fragment Y, then a CORRECT message is broadcast through X to make the fragment identity, level, and state variables of the nodes in X consistent with the variables of the nodes in Y (see section 4.4.1).

- UPDATE

  The root r of fragment X broadcasts an UPDATE message to each node x in X to tell x to enter the UPDATE state (see section 4.6.1).

- U_TEST($FID(X), L(X)$)

  The U_TEST message is a special version of the TEST message used during the Tree Update phase of the algorithm (see section 4.6.1).

- REQUEST

  If a node x executing the Tree Update phase of our algorithm wishes to initiate the cycle resolution algorithm, then x must request the right to initiate the algorithm by issuing a REQUEST message (see section 4.6.2).

- GRANT

  If node x is executing the Tree Update phase and x possesses the right to initiate the cycle resolution algorithm, then x may transfer that right to node y by sending a GRANT message to y (see section 4.6.2).

- CRA((y, z))

  The cycle resolution algorithm removes the heaviest link from a cycle which forms in a fragment. We use the CRA message to find the heaviest link in a cycle. Node u sends CRA((y, z)) if (y, z) is the heaviest link in the path from u to the initiator of the CRA (see sections 4.6.2 and 4.6.3).

- DELETE((y, z)))

  The cycle resolution algorithm issues a DELETE message to tell the nodes in a cycle to remove the heaviest link (y, z) from the cycle (see section 4.6.3).

- CHANGE_PARENT

  After the cycle resolution algorithm removes the heaviest link from the cycle, the algorithm uses the CHANGE_PARENT message to update some parent child relationships (see section 4.6.3).

- RESTART

  If node x in fragment X is executing the Tree Update phase and x wants to re-enter the Tree Construction phase, then x requests the right to restart the Tree Construction phase by sending a **RESTART** message to the root of fragment X (see section 4.6.4).

- ROOT_RESTART

  If the root r of a fragment X receives a **RESTART** message, then r may tell every node in X to restart the Tree Construction phase by broadcasting a **ROOT_RESTART** message through X (see section 4.6.4).

4.2 Tree Construction Phase

Each node x moves between four states: SLEEP, FIND, FOUND, and UPDATE. Initially, x is in the dormant SLEEP state. At some point, x either spontaneously begins execution of the algorithm, or x is triggered to start execution when x receives a message pertaining to the algorithm. Node x exits the SLEEP state and begins execution of the Tree Construction phase of our algorithm by entering the FIND state.

During the Tree Construction phase, each node x moves between the FIND state, in which x helps to find the minimum outgoing link from fragment X, and the FOUND state, in which x helps X merge with another fragment over the minimum outgoing link from X. During the Tree Construction phase, each node x acts to accomplish goals for its fragment X; therefore we say that fragment X moves between the FIND and the FOUND state with the understanding that each node x in X moves between the FIND and the FOUND states with X. The state of a fragment is the state of its root node.

While in the FIND state, fragment X searches for its lightest reliable (not faulty) outgoing link, (x, y). After X identifies (x, y), X enters the FOUND state and X connects with the fragment Y at the other end of (x, y). Eventually, the new fragment Z formed by the merger of X and Y broadcasts an **INITIATE** message to each node z which is a member of Z. When z receives an **INITIATE** message, z enters the FIND state. The cycle of finding the minimum outgoing link (FIND state) and fragment merging (FOUND state) repeats until one fragment W incorporates all n nodes. The final fragment W defines a minimum spanning tree over all the REJECT links. Fragment W enters the UPDATE state when W believes no other fragment exists; this is the beginning of the Tree Update phase of our algorithm.

A new fragment can be formed in one of two ways during the Tree Construction phase, either by the awakening of a single node from the SLEEP state or by the formation of a level L fragment by the merger of two level $L-1$ fragments having the same minimum outgoing link. A single node fragment merely enters the FIND state directly, but the multinode fragment case is more complex.

Suppose a new fragment Z forms by the merger of two fragments X and Y over a link (x, y). Without loss of generality, assume y has the larger ID, $ID(y) > ID(x)$. Then y becomes the root of Z, and $ID(y)$ becomes the FID of Z ($FID(Z) = ID(y)$). The new root y forces fragment Z into the FIND state by broadcasting an **INITIATE**$(FID(Z), L(Z))$ message to the nodes in Z. The two arguments carried by the **INITIATE**$(FID(Z), L(Z))$ message are the level $L(Z)$ and fragment identifier $FID(Z)$ of the new fragment Z. Upon

receiving the INITIATE message, each node z in Z updates its level and FID, and z enters the FIND state after passing the INITIATE message on to its children.

4.3 FIND state: Finding the Minimum Outgoing Link

The first difference between our tree construction algorithm and the GHS algorithm is the process that a node x in the FIND state uses to identify its minimum outgoing link. In the GHS algorithm, x issues a TEST message along only one link at a time when searching for a minimum outgoing link. Because our algorithm must tolerate up to f faulty links (over which a test message may be lost), x issues test messages along $f+1$ links during a minimum outgoing link search.

When a node x begins our algorithm, x assumes each link (x,y) incident to x is an outgoing BRANCH link (i.e., x in fragment X assumes (x,y) connects x to a node y in a different fragment Y). When x enters the FIND state, x starts looking for its minimum weight reliable outgoing link. Node x verifies that the minimum weight BRANCH link (x,y) is actually an outgoing BRANCH link and not an internal REJECT link by sending a TEST$(FID(X), L(X))$ message along (x,y). The TEST message carries two arguments: the fragment identity $FID(X)$ and level $L(X)$ of the fragment X to which x belongs.

Upon receiving a TEST message from x, node y compares its fragment identity $FID(Y)$ with $FID(X)$ in the TEST message. If $FID(Y) = FID(X)$, then x and y are in the same fragment, and y sends a REJECT message back to x. Both x and y classify (x,y) as a REJECT link.

If y receives a TEST$(FID(X), L(X))$ message over link (x,y) and $FID(X) \neq FID(Y)$, then y responds according to the relation between $L(Y)$ and $L(X)$. A fragment X with level $L(X)$ may merge with a fragment Y with level $L(Y)$ if Y is at the end of the minimum outgoing link from X, and the merger obeys the following rules:

1. If $L(Y) < L(X)$, then the merger is delayed until $L(Y) \geq L(X)$.

2. If $L(Y) \geq L(X)$, and X and Y don't share a common minimum outgoing link, then Y absorbs X, that is, X becomes part of Y.

3. If $L(X) = L(Y)$ and X and Y share the same minimum outgoing link (x,y), then X and Y combine into a new fragment Z with new level $L(Z) = L(X) + 1 = L(Y) + 1$. Node y, the node incident to link (x,y) with the larger ID ($ID(y) > ID(X)$), becomes the root of Z.

These rules ensure that a large fragment X does not combine with a smaller fragment Y. Keeping fragments as small as possible for as long as possible reduces the number of messages our algorithm uses to find minimum outgoing links. Node y responds to the TEST message in a way that ensures the fragment merging rules are obeyed. If $L(Y) \geq L(X)$, then y sends an ACCEPT$(FID(Y), FID(X))$ message in response to the test from x. Otherwise, y sends a WAIT message to inform x that the link (x,y) is reliable and that y will send an ACCEPT$(FID(Y))$ or REJECT message when $L(Y) \geq L(X)$.

If x knows all the links in the network are reliable as in the GHS algorithm, then x sends a single TEST message on link (x,y) and x waits until x receives either a REJECT message or an

ACCEPT$(FID(Y), FID(X))$ message response on (x,y). If x receives a REJECT message on the minimum weight BRANCH link (x,y) incident to x, then x classifies (x,y) as a REJECT link and x issues a new TEST message on the next lowest weight BRANCH link. If x receives ACCEPT$(FID(Y), FID(X))$ on (x,y), then x knows (x,y) is the minimum weight outgoing link from x.

In our algorithm, the search by x for its minimum outgoing link is complicated by the fact that x does not know which links in the network are faulty, but if x receives a message along a link (x,y), then x knows (x,y) is reliable. Node x may not be able to identify a single reliable BRANCH link when x enters the FIND state. Since up to f of the links incident to x may be faulty, x issues the message TEST$(FID(X), L(X))$ along the $f+1$ lightest BRANCH links incident to x. By issuing TEST messages over $f+1$ links, x ensures that x will receive an ACCEPT or a REJECT message along at least one link even though the other f links may be faulty. If x knows some links are reliable, then x may be able to issue TEST messages fewer than $f+1$ times and still be assured of receiving a response.

To keep a bound on the number of outstanding TEST messages (those TEST messages to which no response has been received), each node x keeps a list of ASSUMED FAULTY (AF) links. If x does not receive a response to a TEST issued along a link (x,y), then x classifies (x,y) as AF. The sum of the number of TEST messages issued by a node x and the number of AF links incident on x is not allowed to exceed $f+1$. This limits to $f+1$ the number of outstanding TEST messages issued by a node x.

Consider the case of node x, a leaf node of fragment X. If x has q links in its AF list, then x considers its $(f+1)-q$ lightest BRANCH links $l_1, l_2, \ldots, l_{f+1-q}$. If none of these links are known to be reliable, then x issues TEST messages over every link $l_1, l_2, \ldots, l_{f+1-q}$. If one or more of the links in $l_1, l_2, \ldots, l_{f+1-q}$ are known to be reliable, then x issues a TEST message only along the lightest known reliable link l_i and each link with weight less than the weight of l_i.

Node x considers a link (x,y) *tested* if x has sent a TEST message on (x,y). After x issues the TEST messages, x waits for responses. Node x handles each response individually:

- Case 1: REJECT

  If x receives a REJECT message along a link (x,y), then x classifies (x,y) as a REJECT link and x issues enough new TEST messages to ensure that x receives a new response along at least one other link. Node x has issued enough new TEST messages if either x has $f+1$ outstanding (i.e., no response yet received) TEST messages, or x has fewer than f outstanding TEST messages and one of the TEST messages was sent over a link known to be reliable. Note that x issues a TEST message along a light link (x,y) before x issues a TEST over a heavier link (x,z).

- Case 2: x receives ACCEPT$(FID(Y), FID(X))$ along the lightest link (x,y) that x sent a TEST message over.

  If x receives an ACCEPT$(FID(Y), FID(X))$ response along the lightest BRANCH link, then x has successfully located its minimum weight outgoing link (x,y). Node x sends a message REPORT((x,y)) to its parent. After issuing the REPORT message, x enters the FOUND state.

-

Case 3: x receives an ACCEPT($FID(Y), FID(Z)$) message along a link (x, y) and $FID(X) \neq FID(Z)$.

If $FID(X) \neq FID(Z)$, then the ACCEPT message is a response to a TEST message issued by x in a previous FIND state and the ACCEPT message is out of date. $FID(Z)$ is the FID of the fragment x belonged to in a previous FIND state when x sent y a TEST message. When this happens, x simply classifies (x, y) as a reliable link and x waits for the next TEST response to arrive.

.

Case 4: x receives ACCEPT($FID(Y), FID(X)$) along a link (x, y) that is not the lightest link a TEST message was sent over and x does not know of a reliable BRANCH link (x, z) lighter than (x, y).

If x receives an ACCEPT($FID(Y), FID(X)$) message along a link (x, y) and (x, y) is the lightest reliable BRANCH link incident to x, then x accepts (x, y) as its minimum outgoing link and x assumes every BRANCH link incident to x and lighter than (x, y) is faulty.

If x knows a reliable BRANCH link (x, z) exists that is lighter than (x, y) then x waits for the response to the TEST sent over (x, z) before x selects its minimum outgoing link. If (x, z) is an outgoing link, then (x, z) becomes the minimum outgoing link from x.

After x selects a link (x, y) as its minimum outgoing link, x sends REPORT((x, y)) to its parent and x classifies every BRANCH link incident to x and lighter than (x, y) as an ASSUMED FAULTY (AF) link. Node x now enters the FOUND state.

.

Case 5: WAIT

If x receives a WAIT message on link (x, y), then x classifies (x, y) as a reliable link and x waits to receive an ACCEPT or REJECT message along (x, y) or a lighter link.

.

Case 6: There are f or fewer BRANCH links incident to x and none of these links are known to be reliable by x.

Suppose that when x is in the FIND state, there are b BRANCH links and q AF links incident to x, and $b + q \leq f$. If $b + q \leq f$ and none of the BRANCH links are known to be reliable, then x sends TEST messages along the remaining BRANCH links, x reclassifies the remaining BRANCH links as AF links, and x sends REPORT(∞) to its parent. In this situation, it is possible for x to send TEST messages along all the BRANCH links and not receive a reply because all the BRANCH links are faulty.

For example, if node x is incident to $b = 3$ BRANCH links (x, u),(x, v), and (x, w); and $f = 3$ (i.e., there are at most 3 faulty links in the network), then it is possible that (x, u), (x, v), and (x, w) are all faulty. Node x issues TEST messages over (x, u), (x, v), and (x, w). If x waits for a response to the TEST messages and all the links are faulty, then a response never comes and x deadlocks. Rather than risk deadlock, x classifies (x, u), (x, v), and (x, w) as AF links and x continues with the algorithm. If a link (x, u) is not faulty, then u eventually responds to the TEST sent by x. When x receives the response, x reclassifies (x, u) as an ERROR link and x incorporates (x, u) into the MST during the Tree Update phase.

4.3.1 Assumed Faulty Links

If a node x receives a message along an AF link (x, y), then x removes (x, y) from the AF list, x classifies (x, y) as an ERROR link, and x sends an ERR message over (x, y) to force y to classify (x, y) as an ERROR link. When y receives the ERR response, y classifies (x, y) as an ERROR link. If y is in the FIND state and y has issued a TEST message over (x, y), then y issues new TEST messages as though the ERR message were a REJECT message. Node x ignores TEST messages received over ERROR links.

By introducing the ERR message we ensure that if x classifies a link (x, y) as an ERROR link, then y also classifies (x, y) as an ERROR link. The ERROR links are handled in the Tree Update phase of the algorithm.

Thus each node classifies its incident links into four types.

- A REJECT link was tested and was found to connect nodes in the same fragment.
- An ASSUMED FAULTY link did not yield a response to a TEST message.
- An ERROR link was an AF link before a message was received over it.
- All other links are BRANCH links.

4.4 FOUND state: Fragment Merging

Every node v carries out the algorithm described in section 4.3 to locate its minimum outgoing link, but if v is not a leaf node of fragment X, then v waits for REPORT messages from each of its children before v sends a REPORT message to its parent. Node v selects the lightest link (x, y) from among the links that the childrens' REPORT messages cite and the minimum outgoing link from v itself. Node v records the path to the link (x, y) and v sends REPORT$((x, y))$ to its parent. In other words, v reports to its parent the lightest link incident to either v or a descendant of v. After sending the REPORT message, v enters the FOUND state. In this way, the root node r is eventually able to select the fragment's minimum outgoing link (x, y).

After r selects the minimum outgoing link (x, y) from X, r enters the FOUND state and r initiates the change root algorithm. The change root algorithm merges X with Y (the fragment at the other end of (x, y)) by re-ordering the parent pointers in fragment X so that X becomes a subtree of Y, and X is connected to Y along link (x, y).

The root r initiates the change root algorithm by sending a CHANGE_ROOT message to the child u from which r received the REPORT message citing (x, y), and r makes u its new parent. When a node u receives a CHANGE_ROOT message, u sends CHANGE_ROOT along (u, v) leading to the minimum outgoing link (x, y) of the fragment X, and u makes v its new parent. In this way, a CHANGE_ROOT message is sent over each link on the path from the root to the minimum outgoing link (x, y) from fragment X. Eventually node x, adjacent to link (x, y), receives a CHANGE_ROOT message. Node x makes y its new parent, and x sends a CONNECT$(FID(X), L(X))$ message to y along (x, y) ($L(X)$ is the level of fragment X).

When y receives a CONNECT message over (x, y), node y adopts x as a child, and y classifies (x, y) as a REJECT link. Fragment X is now a part of Y.

4.4.1 Response to CONNECT messages

When y receives a CONNECT$(FID(X), L(X))$ message from x along (x, y), node y adopts x as a child, and y classifies (x, y) as a REJECT link. If y is in the FIND state and $L(Y) = L(X)$, then y treats the CONNECT message as an ACCEPT message in that y reports (x, y) as the minimum weight outgoing link from y if (x, y) is lighter than the BRANCH links incident to y and the descendants of y. If $L(Y) > L(X)$, then y treats a CONNECT message like a REJECT message. Node y may carry out other actions depending on the situation:

- Case 1: $L(X) < L(Y)$

  We must update the FID and level of X so that each node in X responds correctly to TEST messages.

  If $L(X) < L(Y)$ when y receives a CONNECT message from x, then y broadcasts a CORRECT$(FID(Y), L(Y), STATE(y))$ message to the nodes that belonged to fragment X ($STATE(Y)$ is the current state of node y: FIND, FOUND, or UPDATE). When a node z receives a CORRECT$(FID(Y), L(Y), STATE(Y))$ message, z updates its FID, level, and state to $FID(Y)$, $L(Y)$, and $STATE(Y)$ respectively, and z broadcasts the CORRECT message to its children.

- Case 2: $L(X) = L(Y)$

  We follow the convention adopted by the GHS algorithm [7] that the fragment identity (of X in this case) changes only when the fragment level changes. Therefore, if $L(X) = L(Y)$, then no extra actions are taken by y until either fragment Y merges with fragment X over (x, y) (i.e., y sends and receives a CONNECT message over (x, y) as described below), or y receives an INITIATE message which y passes on to x (i.e., fragment Y merged with some other fragment Z).

  If $L(X) = L(Y)$ and (x, y) is the minimum outgoing link of both fragments X and Y, then X and Y merge with each other along (x, y) to form a new fragment Z. If node y sends and receives CONNECT messages along the same link (x, y), then y knows fragments X and Y have merged with each other along (x, y) to form a new fragment Z.

  If $ID(Y) > ID(X)$, then y becomes the root of Z. Node y adopts x as a child, and y broadcasts an INITIATE$(FID(Z), L(Z))$ message through the new fragment Z to update the FID and level of the nodes in Z and to force the nodes in Z to enter the FIND state. Fragment Z has root y, fragment ID $FID(Z) = ID(y)$, and level $L(Z) = L(X) + 1 = L(Y) + 1$.

  If $ID(Y) < ID(X)$, then y makes x the parent of y, and y simply idles in the FOUND state until y receives an INITIATE message.

4.5 Summary of Tree Construction Phase

The cycle between FOUND and FIND states continues as the fragments merge into a spanning tree which corresponds to the MST over the REJECT links (i.e., the network minus ASSUMED FAULTY and ERROR links). A fragment root learns that such a tree has been constructed when a search for a minimum outgoing link reveals that no BRANCH links are incident to nodes in the fragment. Our algorithm now enters Tree Update phase to incorporate ERROR links into the MST.

4.6 Tree Update Phase

If a node x receives a message along an ASSUMED FAULTY link (x, y), then x classifies (x, y) as an ERROR link. Each node x ignores ERROR links while in the Tree Construction phase of our algorithm. Our algorithm incorporates ERROR links into the MST construction process during the Tree Update phase.

There are two types of ERROR links: internal ERROR links that join nodes in the same fragment, and outgoing ERROR links that join nodes in different fragments. Our algorithm reclassifies each outgoing ERROR link as a reliable BRANCH link and executes a *cycle resolution algorithm* on internal ERROR links. After executing the cycle resolution algorithm on each internal ERROR link, our algorithm re-enters Tree Construction phase.

4.6.1 Entering the Tree Update Phase

A fragment X enters the Tree Update phase when there are no BRANCH links incident on any node in X (i.e., all the links in X are REJECT, ERROR, or AF links). If there are no BRANCH links in X, then X can't merge with other fragments because X selects its minimum outgoing link from among the BRANCH links.

If the root r of a fragment X determines that there is not an outgoing link from X (i.e., there are no BRANCH links in X), then r tells each node in X to enter the Update state by broadcasting an UPDATE message. When a node x receives UPDATE, x enters the UPDATE state, and x sets its *TOKEN_HOLDER* variable to *parent*. The root r sets its *TOKEN_HOLDER* variable to *self*. We explain the significance of the *TOKEN_HOLDER* variable in section 4.6.2.

The first step of the Update phase is to classify each ERROR link as either internal or outgoing. When node x begins the Tree Update phase, x sends a U_TEST messages along each ERROR link (x, y) incident to x that has not had a REJECT message sent over it and where $ID(x) > ID(y)$. The REJECT message indicates that (x, y) is internal, and we ensure that only one U_TEST message per link is is issued by adding the $ID(x) > ID(y)$ condition. Nodes x and y use the response to the U_TEST message to classify (x, y) as internal or outgoing.

Each node y that receives a U_TEST message over an ERROR link (x, y) waits until y enters the UPDATE state before y responds to the U_TEST message. By having y respond to a U_TEST while y is in the UPDATE state, we avoid the situation where node x classifies (x, y) as a reliable BRANCH link while node y classifies (x, y) as an ERROR link. This situation where x and y do not both classify a link (x, y) as an ERROR link is dangerous because the potential exists for fragment X and fragment Y to merge with each other over different links and become deadlocked.

For example, suppose fragments X and Y are at the same level $L(X) = L(Y)$ and are connected by two ERROR links (x, y) and (x', y') where (x, y) is lighter than (x', y') and (x, y) is much slower than (x', y'). Now suppose fragment X is in the Tree Update phase and Y is in the FIND state of the Tree Construction phase. Since (x, y) is much slower than (x', y'), it is possible for nodes x' and y' to reclassify link (x', y') as a reliable BRANCH link before nodes x and y reclassify (x, y) as a reliable BRANCH link. Fragment Y may merge with fragment X along link (x', y') if (x', y') is the minimum weight BRANCH link incident

to Y. Nodes x and y eventually reclassify (x, y) as a reliable BRANCH link. If (x, y) is the minimum weight BRANCH link incident to X, then X re-enters the Tree Construction phase and X merges with Y along link (x, y). In summary, Y merges with fragment X (making the root of X the root of Y) along link (x', y') and X merges with Y (making the root of Y the root of X) along link (x, y). When this happens, a cycle exists between X and Y and X and Y are deadlocked.

When y is in the UPDATE state, y responds to a U_TEST message like a regular TEST message received over a non-ERROR link except that y classifies (x, y) as a BRANCH, REJECT, or ERROR link based on y's own response to the U_TEST from x (i.e., y does not issue a new U_TEST message over (x, y)). If x receives a REJECT response to a U_TEST issued over (x, y), then x classifies (x, y) as an internal ERROR link. If x receives an ACCEPT response to a U_TEST issued over (x, y), then x classifies (x, y) as an outgoing ERROR link. If x receives a WAIT response, then x waits until x receives an ACCEPT or REJECT response before x classifies (x, y) as internal or outgoing.

4.6.2 Requesting the Cycle Resolution Algorithm

If an algorithm removes the heaviest link from a cycle so the cycle no longer exists, then we say the algorithm *resolves the cycle*. Our algorithm uses a cycle resolution algorithm (CRA) to make the MST defined by a fragment X valid over the union of an ERROR link (x, y) with the current set of REJECT links. The CRA adds (x, y) to X and resolves the resulting cycle (see section 4.6.3). We must execute a CRA on (x, y) to maintain our algorithm invariant that the MST defined by a fragment X is valid over the set of REJECT links in X.

The CRA finds the heaviest link in the cycle formed by adding a link (x, y) to the MST by sending CRA(MAX) messages along the paths from x and y to their least common ancestor. The MAX parameter records the heaviest link a CRA message is sent over. When the least common ancestor (LCA) of x and y receives the CRA messages from x and y, the LCA identifies the heaviest link (u, v) in the cycle by comparing the MAX fields of the two CRA messages. The LCA initiates a process that removes (u, v) from the cycle by sending DELETE(u, v) messages back along the paths to x and y.

The CRA is executed on only one link at a time because each execution of CRA changes the MST topology. Therefore, our algorithm executes CRA on each internal ERROR link sequentially (i.e., one link at a time) by passing an ENABLE token between nodes which request the right to execute the CRA. We ensure that the CRA executes on only one link at a time by treating the CRA as a critical section and protecting it with Raymond's mutual exclusion algorithm [15].

Raymond's algorithm is based on passing a token around a tree. If a node wishes to execute a critical section, then the node must obtain the token from the token's current owner. In our algorithm, a fragment defines a tree, and a node may initiate the CRA only when the node possesses the token.

When a fragment X enters the UPDATE state, the root of X creates a token. Each node x in X maintains a *TOKEN_HOLDER(x)* variable which indicates the position of the token relative to x by recording the ID of the node (either a parent or child of x) which begins the path in X from x to the current location of the token. If x possesses the token, then x sets *TOKEN_HOLDER(x)* to $self$. When the Tree Update phase begins, the root possesses

the token. During the Tree Update phase, each node x keeps the following invariant for its *TOKEN_HOLDER(x)* variable:

- If $TOKEN_HOLDER(x) = w$, then w is the first node on the path from x to the node that possesses the token.

When x possesses the token, x has the right to initiate the CRA on link (x, y) if $ID(X) < ID(Y)$. If x is incident to an internal ERROR link (x, y) and $ID(x) < ID(y)$, then x requests possession of the token so x may initiate the CRA on (x, y). Node x places $self$ at the front of its FIFO queue *REQUEST_Q(x)*. If $ASKED(x) = FALSE$ (i.e., x has not already sent a **REQUEST** message), then x sends a **REQUEST** message to w (the node specified by the *TOKEN_HOLDER(x)* variable) and x sets *ASKED(x)* to *TRUE*.

If w receives a **REQUEST** message from a neighbor u, then w adds u to *REQUEST_Q(w)*. If $ASKED = FALSE$ (i.e., w has not already sent a **REQUEST** message), then w sends a **REQUEST** message to the node cited by *TOKEN_HOLDER(w)*.

If a **REQUEST** message is sent by a node, then the node z which possesses the token will eventually receive a **REQUEST** message [15]. If z receives a **REQUEST** message from a neighbor u, then z enqueues u to *REQUEST_Q(z)*. After z completes execution of the CRA, z passes the token to the node v at the front of *REQUEST_Q(z)* by sending a **GRANT** message to v. After sending the **GRANT**, z removes v from the *REQUEST_Q(z)* , z sets *TOKEN_HOLDER* to v, and z sets *ASKED(z)* to $FALSE$. If the *REQUEST_Q(z)* is not empty, then z sends a **REQUEST** message to the *TOKEN_HOLDER(z)*, and z sets *ASKED(z)* to $TRUE$.

A node w receives the token when w receives a **GRANT** message. Node w sets *TOKEN_HOLDER(w)* to $self$. If $self$ is at the front of *REQUEST_Q(w)*, then w removes $self$ from *REQUEST_Q(w)* and w initiates the CRA.

It is important to note that the token is passed along the tree defined over fragment X at the beginning of the execution of the Tree Update phase. The topology of the tree may change during the execution of the Tree Update phase, but the token is still passed along the links of the original tree. By using the original tree, we save ourselves the trouble of updating *TOKEN_HOLDER* variables for a changing tree.

4.6.3 Cycle Resolution Algorithm

Because every node that requests the token eventually receives the token [15], each node x eventually initiates the cycle resolution algorithm on each incident internal ERROR link (x, y) where $ID(x) < ID(y)$. The CRA inserts (x, y) into the fragment tree X and removes the heaviest link from the resulting cycle [16].

When x receives the token, x initiates the CRA on (x, y) by sending two **CRA**(MAX) messages: one **CRA** message to y and another to the parent of x. Node x classifies (x, y) as a REJECT link and x adopts y as a child. When y receives the **CRA**(x, y) message over (x, y), node y also classifies (x, y) as a REJECT link and y adopts x as a child. When x and y make (x, y) a REJECT link and adopt each other as children, they add (x, y) to their fragment X, forming a single cycle in X.

Each node v that receives a **CRA**(MAX) message along a link (v, w) sends a **CRA**(MAX') message to the parent of v where MAX' is the heavier of MAX and (v, w). In this way the

least common ancestor (LCA) t of x and y eventually receives two CRA(MAX) messages (one from an ancestor of x and the other from an ancestor of y). The ancestors of the LCA receive both CRA messages along the same link and discard their records of the CRA when they receive the second CRA message.

The LCA determines the heaviest link (v, w) in the cycle by comparing the MAX fields of the two CRA messages. The LCA tells the nodes in the MST cycle to remove (v, w) from the X, re-order their parent child relationships, and eliminate their records of the CRA(MAX) message by issuing DELETE(v, w) messages along the paths from the LCA to x and to y.

When a node u not incident to (v, w) and not x (the token holder) receives a DELETE(v, w) message, u sends a DELETE(v, w) message along the link from which u received CRA, and u discards its record of the CRA.

When v (incident to link (v, w)) receives a DELETE(v, w) message, v removes (v, w) from the MST, and v triggers the nodes along the path from v to x to reverse their parent child relationships. Node v disowns its child w, v sends DELETE(v, w) over (v, w) to w, and w discards its record of the CRA.

When w receives a DELETE(v, w) message over (v, w), w disowns its parent v, and w makes the child z from which w received the CRA message the new parent of w. The MAX link (v, w) is no longer a member of the fragment because v disowned its child w and w disowned its parent v. Node w sends a CHANGE_PARENT message to its new parent z.

CHANGE_PARENT messages propagates toward the token holder x reversing parent - child relationships as they go. When a node z receives a CHANGE_PARENT message, z makes its current parent a child, and z makes the node from which z received the CRA(MAX) message the new parent of z.

If node x (the token holder) receives a CHANGE_PARENT message from its parent, then x makes y the new parent of x. If x receives a CHANGE_PARENT message from y, then x ignores the message.

Node x knows the CRA execution on (x, y) is complete when x receives a DELETE(MAX) or a CHANGE_PARENT message from both y and the parent of x. Node x may initiate the CRA on only one link at a time, and x must keep possession of the token while the CRA is executing.

4.6.4 Re-entering the Tree Construction Phase

Each node x reclassifies each outgoing ERROR link incident to x as a reliable BRANCH link during the Tree Update phase. If x is incident to a BRANCH link (x, y), then x requests its fragment X to re-enter the Tree Construction phase by sending a RESTART message to the root r of X. By re-entering the Tree Construction phase, fragment X may merge with fragment Y at the end of (x, y). When re-entering Tree Construction phase, X must take care that no node in X is executing the CRA. When the root r receives a RESTART message, r requests the CRA token (as in section 4.6.2). When r obtains the token, r tells the fragment to re-enter the Tree Construction phase by broadcasting a ROOT_RESTART message.

If x is incident to a BRANCH link, then x sends a RESTART message to its parent. If a node v in the UPDATE state which has neither sent nor received a RESTART or ROOT_RESTART message receives a RESTART message, then v sends a RESTART message

to its parent. If v is not in the UPDATE state or v has already sent or received a RESTART or ROOT_RESTART message, then v ignores new RESTART messages. If a RESTART message is generated, then the root r will eventually receive a RESTART message.

When r receives a RESTART message, r requests the token and r places *restart* at the front of its *REQUEST_Q*. The root r will eventually take possession of the token. If r already has the token, then r waits until the current execution of the CRA is complete. When the CRA is complete, r broadcasts a ROOT_RESTART message which forces X to re-enter the Tree Construction phase.

If a node x in the UPDATE state receives an ROOT_RESTART message, then x waits for the results of the U_TEST messages that x sent over each incident ERROR link. After receiving the result for each U_TEST message, x empties its *REQUEST_Q*, and x enters the FIND state. Node x uses the results of the U_TEST messages to reclassify each ERROR link as either a reliable BRANCH link or an internal ERROR link. Rather than issuing new TEST messages in the FIND state, x uses the results of the U_TEST messages to identify its minimum outgoing link.

4.6.5 Termination of the Algorithm

Eventually a single fragment X forms on the network (theorem 3). When X can not find an outgoing link, X enters the UPDATE state. Since no outgoing ERROR links exist, X remains in the UPDATE state and executes the CRA on each internal ERROR link. An MST over the reliable links of the network exists when no ERROR links exist in X (i.e., the CRA has resolved all the internal ERROR link cycles). At this point every link is either a REJECT link or an AF link.

5 Correctness

If our algorithm constructs a minimum spanning tree, then our algorithm is correct. Our proof of correctness has three parts. First, we show that our algorithm constructs a minimum spanning tree over the set of REJECT links belonging to a fragment. Then we show that every reliable link is eventually classified as a REJECT link. Last, we show that a single fragment eventually forms over all the nodes of the network.

5.1 Preliminary Lemmas

Lemma 1 *A minimum spanning tree may be constructed on a graph with uniquely weighted links by joining each fragment X with another fragment Y along the minimum weight outgoing link from X to form a new fragments and iterating [6].*

Lemma 2 *Let (V,E) be a tree where V is the set of vertices and E is the set of links. The MST over (V,E ∪ E') where E' is a set of links disjoint from E but connecting nodes in V, may be constructed by adding a link from E' to the tree (V,E), resolving the resulting cycle, and iterating over all the edges in E' [16].*

Lemma 3 *Each link (x, y) incident to a node x that participates in our algorithm has a* TEST *message sent over it.*

PROOF: When a node x begins executing our algorithm, x classifies each link (x, y) incident to x as a BRANCH link that is not known to be reliable, and x enters the Tree Construction phase. Node x remains in the Tree Construction phase until no BRANCH links are incident to x, and x does not reclassify a link (x, y) unless x sends or receives a TEST message over (x, y). Therefore, node x eventually sends or receives a TEST message over (x, y). □

Lemma 4 *Every node in the network participates in our algorithm.*

PROOF: Before our algorithm begins execution, every node is in the SLEEP state. We assume that at least one node x spontaneously leaves the SLEEP state and begins execution of our algorithm. By lemma 3, a TEST message is eventually sent over each link (x, y) incident to x.

When a sleeping node y receives a TEST message, y leaves the SLEEP state and begins executing our algorithm. By lemma 3, a TEST message is eventually sent over each link (y, z) incident to y.

Since the network of reliable links is connected, every node eventually leaves the SLEEP state and begins executing our algorithm. □

Lemma 5 *At least one* TEST *message is eventually sent over every link* (x, y).

PROOF: Every node participates in our algorithm (lemma 4) and every link (x, y) incident to a node x that participates in our algorithm has a TEST message sent over it (lemma 3). Therefore, at least one TEST message is eventually sent over every link. □

5.2 Theorems Showing Correctness

Theorem 1 *When no node is executing the cycle resolution algorithm, each fragment X defines the minimum spanning tree over the REJECT links incident to nodes in X.*

PROOF: Let R be the set of REJECT links incident to the nodes of a fragment X at an arbitrary time when no node is executing the cycle resolution algorithm. A link (x, y) may become a member of R in one of three ways. In each of these cases, the fragment maintains the MST over R.

First, if node x in fragment X sends a CONNECT message over link (x, y) to node y in fragment Y, then x and y classify (x, y) as a REJECT link and (x, y) becomes part of the MST defined by the fragment containing x and y. Since x sent a CONNECT message over (x, y), we know (x, y) was the minimum weight outgoing link from fragment X over the set of known reliable links. Therefore, (x, y) belongs to the minimum spanning tree by lemma 1.

Second, if node y sends a REJECT message over (x, y) in response to a TEST from x and (x, y) is not an AF link, then x and y classify (x, y) as a REJECT link that is not part of the MST. Fragments X and Y had already merged along a link (u, v) that is lighter than (x, y). Therefore, (x, y) does not belong in the MST.

Third, if (x, y) is an internal ERROR link and $ID(X) < ID(Y)$, then during the Tree Update phase, x initiates the cycle resolution algorithm on (x, y). The cycle resolution

algorithm resolves the cycle formed by adding (x, y) to the MST; by lemma 2 the MST is maintained. □

Theorem 2 *Every reliable link (x, y) is eventually classified as a REJECT link by both x and y.*

PROOF: A node x classifies every incident link (x, y) as being one of five types:

1. a BRANCH link not known to be reliable
2. a reliable BRANCH link
3. a REJECT link
4. an ASSUMED FAULTY link
5. an ERROR link.

At the beginning of our algorithm, each node x classifies every link (x, y) incident to x as a BRANCH link that is not known to be reliable. By lemma 5, a TEST message is eventually sent over link (x, y). Node x classifies (x, y) as a reliable BRANCH link, a REJECT link, or an ASSUMED FAULTY (AF) link based on the response or lack of response to the TEST message sent over (x, y).

If x sends a TEST message over (x, y) and x believes link (x, y) is faulty (because the response to the TEST is late or there are fewer than f faulty links incident to x), then x classifies (x, y) as an AF link. If (x, y) is actually reliable, then x eventually receives a response to the TEST x sent over (x, y). When x receives a message over an AF link (x, y), x reclassifies (x, y) as an ERROR link.

During the Tree Update phase, x reclassifies each ERROR link (x, y) as either a reliable BRANCH link (if (x, y) is an outgoing link) or a REJECT link (if (x, y) is an internal link).

Let (x, y) be a reliable BRANCH link connecting fragment X (containing x) with fragment Y (containing y). If (x, y) is the lightest link connecting X and Y, then x eventually sends a CONNECT message over (x, y), and (x, y) becomes a REJECT link that is part of the MST. If (x, y) is not the lightest link connecting X and Y, then X and Y merge along a lighter link (u, v), and x eventually sends or receives a REJECT message over (x, y), making (x, y) a REJECT link that is not part of the MST. □

Theorem 3 *Every node of the network eventually belongs to a single fragment.*

PROOF: Every reliable link (x, y) eventually becomes a REJECT link (theorem 2). Since the network is connected and REJECT links connect nodes in the same fragment, every node of the network must eventually belong to the same fragment. □

6 Message Complexity

We present here an upper bound on the number of messages exchanged during the execution of our algorithm. Note that our most complex message contains two unique identifiers (of links, fragments, or levels) and a few bits to indicate the message type.

Each reliable link becomes a REJECT link exactly once.

- Two messages (TEST and REJECT) are required to reject a link (x, y), and both x and y must reject (x, y).

This yields a maximum of $4e$ messages for REJECT link classification.

Each node x classifies at most f links as AF links. Since each AF link may become an ERROR link, at most fn links may become ERROR links. A link can be classified as an ERROR link at most once (i.e., once an ERROR link (x, y) is reclassified as a BRANCH or REJECT link, (x, y) can not become an ERROR link again). Error links generate most of the overhead in our algorithm. For each error link (x, y):

- at most 4 messages are used to classify (x, y) as an ERROR link (a TEST from x and a response from y, and a TEST from y and a response from x),
- at most 2 messages are used to classify each ERROR link (x, y) as internal or outgoing (a U_TEST from y and a response from x),
- at most n REPORT and n UPDATE messages are generated to enter the Tree Update phase,
- at most n REQUEST and n GRANT messages are generated to move the CRA token,
- at most $2n$ CRA messages are generated to identify the heaviest link in the cycle,
- at most n DELETE and CHANGE_PARENT messages are generated to remove the heaviest link from the cycle,
- at most n RESTART and n ROOT_RESTART messages are generated to re-enter the Tree Construction phase.

A maximum of $fn(9n + 6)$ messages are devoted to ERROR links.

Since $\log_2 n$ is an upper bound on the fragment level, each node can go through at most $-1 + \log_2 n$ levels not counting the zeroth and last level. At each intermediate level, each node can:

- receive at most 1 INITIATE or CORRECT messages,
- receive at most $f + 1$ ACCEPT messages in response to at most $f + 1$ TEST messages,
- send at most 1 REPORT message,
- send at most 1 CHANGE_ROOT or CONNECT message,

These messages add at most $(2f+5)n(-1+\log_2 n)$ to our total.

At level zero each node can:

- receive at most 1 CORRECT message,
- receive at most $f+1$ ACCEPT messages in response to at most $f+1$ TEST messages,
- send at most 1 CONNECT message.

The zeroth level adds at most $n(2f+4)$ messages to our algorithm.

The last level each node sends one REPORT message and one UPDATE message adding $2n$ messages to our total.

Finally, each node incident to a faulty link (x, y) sends one TEST message over (x, y). At most $2f$ messages are sent over faulty links.

Summing our subtotals we find that the maximum number of messages sent During the execution of our algorithm is at most:

$$(2f+5)n\log_2 n + n + 4e + fn(8n+6) + 2f$$

. Our message complexity is $O(fn\log n + fn^2)$.

7 Time Complexity

We consider the time complexity of the Tree Construction phase and the Tree Update phase separately. We calculate the overall time complexity of the algorithm by summing the time complexities of the two phases. This is a reasonable thing to do because in the worst case the Tree Construction phase messages are followed sequentially by the Tree Update phase messages.

We first consider the time complexity of the Tree Construcion phase. We make the simplifying assumption that WAKE_UP messages are sent to every node so that every node begins execution of the algorithm within $n-1$ time units of the time at which the initiator begins the algorithm [7]. At each level, the worst case numbers of sequential messages are:

- n sequential TEST messages yielding n sequential responses,
- n sequential REPORT messages,
- n sequential CHANGE_ROOT or CONNECT messages,
- n sequential INITIATE or CORRECT messages.

Since each node goes through at most $1+\log_2(n)$ phases, a maximum of $5n+5n\log_2 n$ sequential messages are sent during the Tree Construction phase assuming a WAKE_UP message is broadcast at the beginning of the algorithm.

The Tree Update phase sequentially executes the CRA on up to fn ERROR links. Each execution of the CRA involves at most:

- n sequential REPORT(∞) messages,

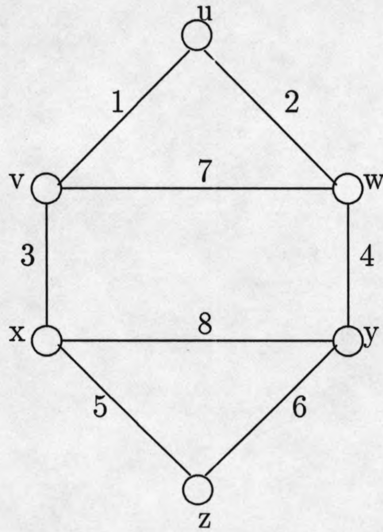
- n sequential UPDATE messages,
- n sequential REQUEST messages,
- n sequential GRANT messages,
- n sequential CRA messages,
- n sequential DELETE or CHANGE_PARENT messages,
- n sequential RESTART messages,
- n sequential ROOT_RESTART messages.

During Tree Update phase our algorithm sends at most $8fn^2$ sequential messages.

The time complexity of our algorithm is $O(n \log n + fn^2)$.

8 Example

Figure A: Our Network



We present here an example of an execution of our algorithm on the six node network shown in *figure A*. We identify nodes by letter (u,v,w,x,y,z) and links by weight ($L1$,$L2$,$L3$,$L4$,$L5$,$L6$,$L7$,$L8$) where link $L1$ has weight 1, $L2$ has weight 2, ... We denote fragments with a series of capital letters corresponding to the nodes in the fragment. For example, fragment UVW consists of nodes u, v, and w.

We assume $f = 1$, that is, there may be at most one faulty link in the network, and we assume $L3$ is faulty. For simplicity we assume that nodes u, v, and w undergo an execution of our algorithm that is identical to the execution of nodes z, x, and y respectively to form

fragments UVW and XYZ. Fragments UVW and XYZ merge to form a complete MST over the network.

8.1 Tree Construction Phase: Formation of Fragment UVW

Node u spontaneously begins execution of our algorithm by entering the FIND state. Node u defines a single node fragment U at level 0. Node u sends $f+1=2$ TEST$(u,0)$ messages, one along $L1$ and another along $L2$. We assume link $L1$ is slow, so node w receives its TEST message from u before node v receives its TEST.

When v and w receive their TEST$(u,0)$ messages, they each leave the SLEEP state and enter the FIND state. Both v and w respond to the TEST message with an ACCEPT message because v and w are each in different fragments than u, and nodes u, v, and w are at the same level 0.

If w did not know whether the $f=1$ lightest link incident to w was reliable, then w would issue $f+1=2$ TEST messages along the two lightest links incident to w. Node w knows $L2$ is reliable because w received the TEST message from u along $L2$. Therefore, w issues a single TEST$(w,0)$ message along link $L2$. Similarly, node v sends a single TEST$(v,0)$ message along $L1$.

Because link $L1$ is slow, u receives the ACCEPT message from w along $L2$ before u receives the ACCEPT(v,u) message from v along link $L1$. Node u selects $L2$ as its minimum outgoing link and classifies $L1$ as an AF link. Now u enters the FOUND state. Node u makes w its parent, classifies $L2$ as a REJECT link, and sends a CONNECT message along $L2$.

When u receives the ACCEPT(v,u) message from v along link $L1$, u classifies $L1$ as an ERROR link, and u sends an ERR message along $L1$ to force v to classify $L1$ as an ERROR link too. Node u ignores the TEST$(v,0)$ message from v along the ERROR link $L1$.

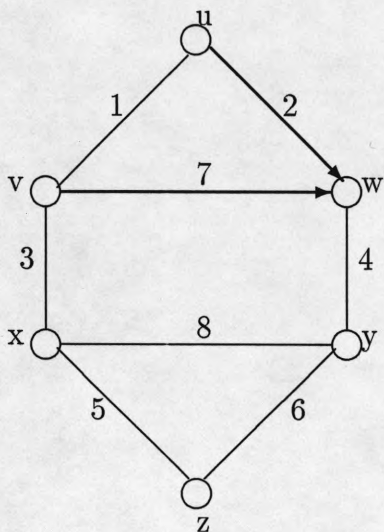
When u receives the TEST$(w,0)$ message from w, node u responds with an ACCEPT(u,w) message.

When v receives the ERR message from u, node v classifies $L1$ as an ERROR link and v issues new TEST$(v,0)$ messages along links $L3$ and $L7$ so that v maintains $f+1=2$ outstanding TEST messages. Node w responds to the TEST$(v,0)$ message with an ACCEPT message. When v receives the ACCEPT(w,v) message from w, node v accepts link $L7$ as the minimum outgoing link from v, and v classifies $L3$ as an AF link. Node v enters the FOUND state, and v sends a CONNECT$(w,0)$ message to w on $L7$.

When w receives the CONNECT$(v,0)$ message from v, node w classifies v as a child and $L7$ as a REJECT link. When w receives the CONNECT message from u along $L2$, w adopts u as a child, and w classifies $L2$ as a REJECT link. Because w has issued a TEST message along $L2$, the fragments containing u and w are both at the same level 0, and w is in the FIND state when w receives the CONNECT message, it follows that w classifies $L2$ as its minimum outgoing link and enters the FOUND state. Node w sends a CONNECT$(w,0)$ message to u along $L2$, and w marks u as the parent of w. Since w and u both send and receive a CONNECT message along $L2$, they merge to form a new fagment. Since $ID(w) > ID(u)$, node w becomes the root of the new fragment UVW. Node w sets its level to $0+1=1$, and sets its fragment identifier to $FID = ID(w)$ (the ID of the new fragment's root w). Node w adopts u and v as children, and w broadcasts an INITIATE$(FID = w, Level = 1)$ message

to its children, u and v. When u and v receive their **INITIATE** messages from w, nodes u and v update their FID and level variables, and they enter the FIND state.

Figure B: Network at beginning of the Tree Update phase



Neither u nor v is incident to a BRANCH link. Therefore, u and v each issue a $\mathsf{REPORT}(\infty)$ message to their mutual parent, w.

Node w is incident to one BRANCH link $L4$ $(1 < f + 1)$, and w does not know wheter $L4$ is reliable or faulty. Therefore, w issues a $\mathsf{TEST}(FID = w, Level = 1)$ message along $L4$, and w classifies $L4$ as an AF link.

Node w is the root of the fragment consisting of nodes u, v, and w (fragment UVW). After w receives the $\mathsf{REPORT}(\infty)$ messages from u and v, the root w enters the FOUND state. When w enters FOUND state, w determines the minimum outgoing link from UVW.

Since no BRANCH links exist in fragment UVW, w broadcasts **UPDATE** messages to its children u and v, and w enters the UPDATE state. Nodes u and v enter the UPDATE state when they receive the **UPDATE** messages from w. We assume that w receives a response to the **TEST** issued over $L4$ shortly after w enters the UPDATE state; w classifies $L4$ as an ERROR link and w sends an **ERR** message over $L4$. *Figure B* depicts the state of the network at the beginning of the Tree Update phase.

8.2 Tree Update Phase and the Cycle Resolution Algorithm

When w enters the UPDATE state, w sets *TOKEN_HOLDER* to $self$ because w is the root. In other words, w possesses the CRA token for fragment UVW at the beginning of the UPDATE phase. Upon entering the UPDATE state, the other nodes in the fragment (u and v) set their *TOKEN_HOLDER* variables to w, their parent.

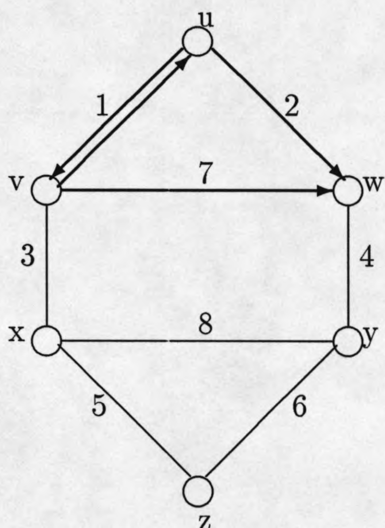
Nodes u and v are incident to ERROR link $L1$. Since $ID(v) > ID(u)$, node v sends a $\mathsf{U_TEST}((FID = w, Level = 1)$ message over $L1$ to determine whether $L1$ is an internal or

outgoing ERROR link. Node u responds to the U_TEST message with a REJECT message, and u classifies $L1$ as an internal ERROR link. When v receives the REJECT message, v also classifies $L1$ as an internal ERROR link. Since $ID(v) > ID(u)$, node v requests the CRA token by sending a REQUEST message to w, the *TOKEN_HOLDER*.

When w receives the REQUEST message from v, node w sends a GRANT message to v, and w sets its *TOKEN_HOLDER* variable to v. When v receives the GRANT message, v sets its *TOKEN_HOLDER* variable to $self$, and v initiates the CRA algorithm on $L1$. Node v classifies $L1$ as a REJECT link, and v adopts u as a child. Node v sends CRA($L1$) messages to nodes u and w over links $L1$ and $L7$ respectively.

When u receives the CRA($L1$) message, u classifies $L1$ as a REJECT link, and u adopts v as a child. A cycle now exists in our network as depicted in *figure C*. Node u sends a CRA($L1$) message to w over link $L2$.

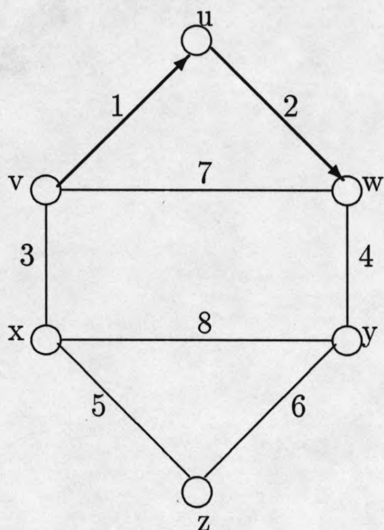
Figure C: Network with an ERROR link cycle



Node w eventually receives two CRA($L1$) messages, one from v over $L7$ and one from u over $L2$. By comparing the weights of $L1$, $L2$, and $L7$, node w determines that $L7$ is the heaviest link in the cycle, and w sends DELETE($L7$) messages to u and v. Since w is incident to $L7$, w removes $L7$ from the MST by disowning its child v.

When node u receives the DELETE($L7$) message, u simply sends a DELETE($L7$) message to v, the node that sent u the CRA message. When v receives the DELETE($L7$) message over $L7$ from w, node v disowns its parent w, and v adopts u as its new parent. Node v knows the CRA is complete when v receives the second DELETE($L7$) message from u along $L1$. *Figure D* illustrates the state of our network at this point in the algorithm.

Figure D: Network after the Tree Update phase



Node w is incident to ERROR link $L4$. Node w sends a U_TEST($FID = w, Level = 1$) message over $L4$ to determine whether $L4$ is an internal or outgoing ERROR link. We assume that nodes x, y, and z form a fragment XYZ via a process analogous to the formation of UVW. Therefore, w eventually receives an ACCEPT response to the U_TEST message and w classifies $L4$ as a reliable BRANCH link some time after w grants v the CRA token.

Since w is now incident to a reliable BRANCH link, w requests that fragment UVW re-enters the Tree Construction phase. If w were not the root, then w would have to send a RESTART message to the root to tell fragment UVW to re-enter the Tree Construction phase. Since w is the root, w does not need to send a RESTART message, but w must obtain the CRA token before w can broadcast a ROOT_RESTART message. Node w sends a REQUEST message to v, the *TOKEN_HOLDER*. When v completes the CRA, v sends a GRANT message to w and v sets its *TOKEN_HOLDER(v)* variable to w. When w receives the GRANT message, w sets *TOKEN_HOLDER(w)* to *self* and w broadcasts a ROOT_RESTART message which tells the nodes in fragment UVW to enter the FIND state.

Since nodes u and v are not incident to any BRANCH links, they both send REPORT(∞) messages to w and enter the FOUND state, but w is incident to $L4$. Node w knows $L4$ is a BRANCH link because w received an ACCEPT response to a U_TEST message sent over $L4$. Therefore, w classifies $L4$ as the minimum weight outgoing link from UVW and w enters the FOUND state.

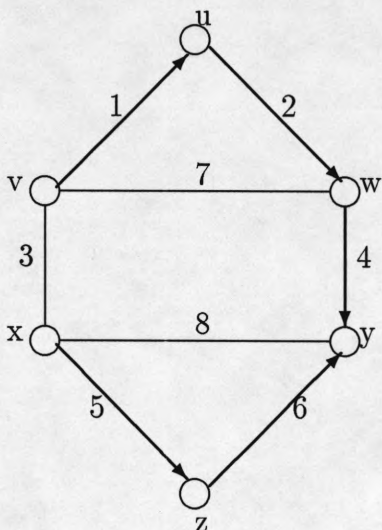
8.3 Completion of the Algorithm

Node w sends a CONNECT message over $L4$, and w makes y its new parent. By a similar process, node y sends a CONNECT message over $L4$. Since $ID(y) > ID(w)$, y becomes the root of the new fragment $UVWXYZ$ formed by merging fragments UVW and XYZ

over link $L4$. Node y sets $Level = Level + 1 = 2$ and $FID = y$ and y broadcasts an INITIATE($FID = y, Level = 2$) message to its children to tell them to enter the FIND state.

Since no BRANCH links remain, fragment $UVWXYZ$ enters the FOUND state then the UPDATE state. No ERROR links exist either; therefore, the algorithm idles in the UPDATE state. This is the end of the execution of the algorithm. Note that every reliable link is classified as a REJECT link and the one faulty link $L3$ is classified as an ASSUMED FAULTY (AF) link. *Figure E* shows the complete MST on our network.

Figure E: Network with complete MST



9 Afterthoughts

Our algorithm is open to a great deal of improvement and variation. We should note these points about our algorithm:

- If $f = 0$, then our algorithm is equivalent to the GHS algorithm.
- If an initially faulty link is repaired, it is easily added to the MST via the Tree Update phase.
- The longest message sent by our algorithm has O(log(max. weight)) message length.

Some problems open for further study are:

- There are methods for reconstructing an MST after link deletion [14]. These may be applicable to modifying our algorithm to allow link failure during execution.

- The time complexity of the GHS algorithm has been reduced in subsequent papers [2, 5, 8]. These algorithms may have beneficial effects when used in place of the GHS algorithm in the Tree Construction phase of our algorithm.

- Several papers deal with fault tolerant spanning tree construction and leader election [1, 4, 9, 10, 13]. It may be possible to apply the concepts developed in these papers to the MST problem in a way that is more direct than the path we have taken.

- The bottleneck of our algorithm occurs in the Tree Update phase. We implement a serial algorithm to solve the cycle resolution problem, but we believe a more efficient parallel algorithm may be developed. We cite two papers on the subject of tree update after topology change [14, 16].

References

[1] H. Abu-Amara, "Fault-Tolerant Distributed Algorithm for Election in Complete Networks," *IEEE Transactions on Computers,* Vol. 37, No. 4, April 1988, pp. 449–453.

[2] B. Awerbuch, "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems," *Proceedings of the 19th Annual ACM Symposium on Theory of Computing,* 1987, pp. 230–240.

[3] B. Awerbuch, I. Cidon, and S. Kutten, "Communication-Optimal Maintenance of Replicated Information," *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,* Vol. 2, 1990, pp. 492–502.

[4] R. Bar-Yehuda and S. Kutten, "Fault Tolerant Distributed Majority Commitment," *Journal of Algorithms,* Vol. 9, 1988, pp. 568–582.

[5] F. Chin and H. F. Ting, "Improving the Time Complexity of Message-Optimal Distributed Algorithms for Minimum-Weight Spanning Trees," *SIAM Journal on Computing,* Vol. 19, No. 4, August 1990, pp. 612–626.

[6] E. A. Estes, "Distributed Minimum Spanning Tree Construction," University of Illinois at Urbana - Champaign, Master's Thesis (ECE Department), 1987.

[7] R. G. Gallager, P. A. Humblet, P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Transactions on Programming Languages and Systems,* Vol. 5, No. 1, January 1983, pp. 66–77.

[8] J. A. Garay, S. Kutten, and D. Peleg, "A Sub-Linear Time Distributed Algorithm for Minimum-Weight Spanning Trees," *Proceedings of the 34th Annual Symposium on Foundations of Computer Science,* 1993, pp. 659–668.

[9] A. Itai, S. Kutten, Y. Wolfstahl, and S. Zaks, "Optimal Distributed t-Resilient Election in Complete Networks," *IEEE Transaction on Software Engineering,* Vol. 16, No. 4, April 1990, pp. 415–420.

[10] A. Itai and M. Rodeh, "The Multi-Tree Approach to Reliability in Distributed Networks," *Information and Computation,* Vol. 79, 1988, pp. 43–59.

[11] E. Korach, S. Moran, and S. Zaks, "Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Network of Processors," *Proceedings of the 3rd Annual ACM Symposium on the Principles of Distributed Computing,* 1984, pp. 199–207.

[12] E. Korach, S. Moran, S. Zaks, "The Optimality of Distributive Constructions of Minimum Weight and Degree Restricted Spanning Trees in a Complete Network of Processors," *SIAM Journal on Computing,* Vol. 16, No. 2, April 1987, pp. 231–236.

[13] S. Kutten, "Optimal Fault-Tolerant Distributed Construction of a Spanning Forest," *Information Processing Letters,* Vol. 27, 1988, pp. 299–307.

[14] J. Park, T. Masuzawa, K. Hagihara, and N. Tokura, "Distributed Algorithms for Reconstructing MST after Topology Change," *Proceedings of the 4th International Workshop on Distributed Algorithms,* Lecture Notes in Computer Science, Vol. 486, Springer Verlag, 1991, pp. 122–132.

[15] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Transactions on Computer Systems,* Vol. 7, No. 1, 1989, pp. 61–77.

[16] H. Tsin, "An Asynchronous Distributed MST Updating Algorithm for Handling Vertex Insertions in Networks," *Proceedings of the International Conference on Parallel Processing and Applications,* 1987, pp.221–226.