BROADCASTING TOPOLOGY AND ROUTING INFORMATION IN COMPUTER NETWORKS

by

John Michael Spinelli

B.E. The Cooper Union
(1983)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1985

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 23, 1985

Certified by _____
Robert G. Gallager
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

BROADCASTING TOPOLOGY AND ROUTING INFORMATION IN COMPUTER NETWORKS

by

John Michael Spinelli

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
May 1985

ABSTRACT

The topology problem in store and forward computer networks is that of keeping all nodes informed of the current operational status of each communication link in the network. The failure or repair of one or more communication links is called a topology change. Two efficient algorithms are presented for solving this problem. They require $O(l)$ communication and $O(n)$ time for simple topology changes in a network with l links and n nodes. The algorithms send messages only in response to topology changes, and each message usually contains information only about the links whose status has changed. The algorithms work properly in the presence of arbitrarily complex topology changes.

The routing information problem is that of keeping each node informed of the the packet transmission delay on each directed link in the network. Nodes need this information in order to make intelligent routing decisions. The two topology algorithms discussed above are used to also solve the routing information problem. The level of difficulty in solving this problem is found to depend greatly on the particular properties of the topology algorithm used.

Thesis Supervisor: Dr. Robert G. Gallager

Title: Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I thank my parents, Nancy and Tom Spinelli, for all their love and encouragement throughout my life.

I thank my advisor, Robert Gallager, for his brilliant insight, his patient guidance, and his unfailing optimism.

I thank Frank Gentile for his friendship, and P. Rochelle Berg for her friendship and her help with the typing.

I thank Jean Régnier for many helpful discussions, and Arthur Giordani for his fine art work.

Special thanks to Tulio.

This work was supported in part by a National Science Foundation Graduate Fellowship, National Science Foundation contract ECS-8310698, and Defense Advanced Research Projects Agency contract N00014-84-K-0357. The funding from each of these sources was greatly appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Computer Network Model

A computer network consists of a set of computers called nodes which are connected by bidirectional communication channels called links. An example of such a network is shown in Figure 1.1.1. The bidirectional arrows at each node indicate that messages enter and leave the network at those points. The function of the network is to accept a message at some source node and to deliver it to the appropriate destination node. The external source which supplies messages to a node may be a computer or a user's terminal and is not significant in our work. We shall view nodes as being both sources and destinations of messages and will not be concerned with how these messages are generated or used. What we are calling a computer network is usually referred to as a "sub-network" in the literature [1].

This thesis will be exclusively concerned with packet switched networks. A packet network is one which divides a long message into bundles, called packets, which are then sent separately from the source node to the destination node. An important job of the network is to decide which of several possible routes a packet traveling from one node to another should take. This is known as the routing problem. For example, in Figure 1.1.1 node 1 could send a packet to node 4 through either node 3 or node 2. At each node processor, a routing algorithm is used to make these decisions. A major goal of this thesis is to provide the routing algorithm at each node with information upon which to base its routing decisions. The specific contents of this information will be discussed shortly.

In order to state the problem precisely, it is useful to introduce several concepts from graph theory [2]. We can view the network of Figure 1.1.1 as an undirected graph with

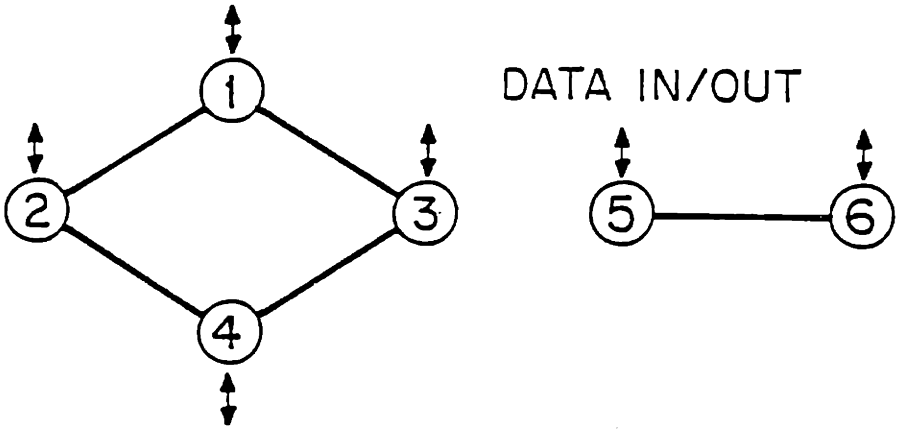

Figure 1.1.1: A Computer Network Model

6 nodes and 5 links (also called arcs and edges). The information contained in this graph, i.e. which links are connected to which nodes, is known as the network topology. A given link is said to be incident on two nodes which are called the end nodes of that link. Two nodes are connected if a path exists between them. A node is said to be connected to a link if a path exists between the node and either of the link's end nodes. A maximum connected set of nodes is a largest possible set of mutually connected nodes. In Figure 1.1.1 there are two maximum connected sets of nodes, {1,2,3,4} and {5,6}. Each of these sets is called a component. A maximum connected set of links is the set of all links which are incident on one or more members of a given maximum connected set of nodes. We will often use the term maximum connected set to refer to either links, nodes, or both.

The number of links and nodes in a network are not independent. It can be shown that in a connected network with n nodes, the number of links, l, must be bounded by:

$$n - 1 \leq l \leq \frac{n(n-1)}{2}$$

Figure 1.1.2 shows two extreme situations for a 4 node network. The left hand network contains the maximum number of links and is called fully connected. The right hand network is as sparsely connected as possible. One of the major motivations for using packet

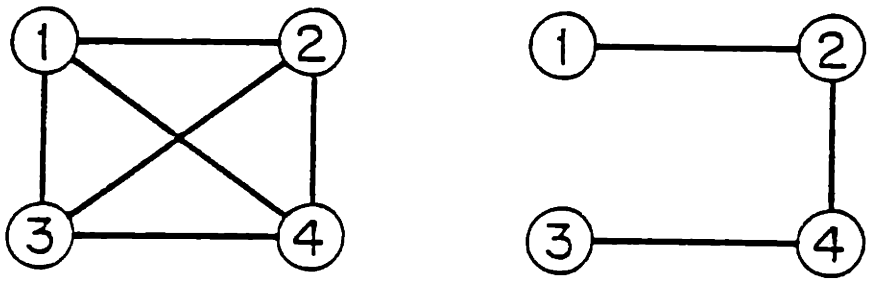

Figure 1.1.2: Connectivity Example

switching networks is to interconnect many nodes without needing a communication channel between each pair. Thus, in practice, the number of links in an n node network is usually much closer to the lower bound than the upper bound. For example, a common computer network, the ARPANET, at one time had 62 nodes and 74 links [4].

So far, we have been discussing general packet switching networks. However, this thesis is concerned with networks that have some additional restrictions. Our work is applicable to wide-area networks with topologies which do not usually change very often. The classic example of this type of network is the ARPANET [3]. The wide area assumption implies that communication over the network is expensive (i.e. a significant fraction of the total cost of the network) and should be minimized. We are concerned with networks where the topology is usually static, but may change occasionally due to a link or node failing or being repaired. This work may be applicable to broadcast networks provided that their topology can be represented by a graph which does not change rapidly. We exclude situations, such as mobile packet radio, where the network topology is in a nearly continuous state of change.

1.2 The Topology Problem

At any time while a network is in operation, one of its links or nodes may malfunction. We call this event a link or node failure. For links, this may occur because the link has been physically damaged or because the error rate on the link has become so high as to make it unusable. Also, at any time a link or node which had failed may become operational again. Each of these failures or repairs is known as a topology change.

It is usually very desirable for each node in a network to be kept aware of the current network topology. A node needs to know which other nodes it is disconnected from so that it will not try to send packets to those nodes. The routing algorithm at each node needs to know which links are not operating so that it can avoid routing packets on them. Unfortunately, due to the inherent communication delay in the network, no node can be sure that it is aware of the present network topology. However, we can design algorithms which guarantee that a finite time after a link topology change occurs, each node connected to that link is made aware of the change. The major focus of this thesis is to design algorithms which have this property, perhaps subject to some additional assumptions and requirements. We refer to this as the topology problem.

Solutions to the topology problem can consider link topology changes to be a general case. When a node failure occurs, we assume that each link incident on that node also fails. When a node is repaired, at least one of its incident links should also begin operating. In this way, all node failures and repairs can be considered to be a set of link topology changes.

We can define the topology problem more precisely by first considering how a link topology change occurs. At each end node of a link, there is a device called a data link controller (DLC). The two DLC's associated with a link are responsible for sending and receiving information over it. When the link is operating, they provide error free communication by means of some protocol protection mechanism. However, at any time, either of a link's DLC's may decide that the link is not operating. We require that both of a link's DLC's eventually reach the same decision about the operating status of a link,

although not necessarily at the exact same time. Each link is either operating (up) or not operating (down) at a given time. We can resolve the conflict during the short interval when the two DLC's may have different opinions by considering the link to be down during that time.

Using this definition of link status, we can define the topology problem precisely.

Topology Problem. *If no status changes have occurred for a sufficient but finite time, each node in a maximum connected set of nodes must know the correct status of each link incident on a member of that set.*

This is typically accomplished by broadcasting topology information using special message packets. We define broadcasting as the process of sending a message to every node in the network which can be reached from a given source node. Several methods of doing this will be discussed later. The requirement that no status changes occur for a sufficient but finite time is meant to exclude situations where the topology is rapidly and continuously changing. Our solutions to the topology problem will work properly even if a large number of changes occur in a short time. However, in order for the algorithms which we design to terminate (stop sending messages and adopt a consistent topology) there must eventually be a stagnant period during which no link status changes occur. This corresponds to placing a constraint on the average time between link status changes in a network, but not placing any restrictions on the short term behavior of a link.

At this point it is useful to examine a simple example which illustrates one of the reasons why the topology problem is difficult to solve (refer to Figure 1.2.1). Assume that link l fails and then, a very short time later, is repaired. Also assume that we adopt the simple rule that when a node finds out about a topology change it sends a message to each of its neighbors notifying them of the change. Thus, node 3 first sends a message to 1 and 2 telling them that link l has failed. A short time later it sends two more messages telling them that l has been repaired. Assume that node 2 receives the two messages from node 3 very quickly and sends them on to node 1. Assume that node 1 receives the two

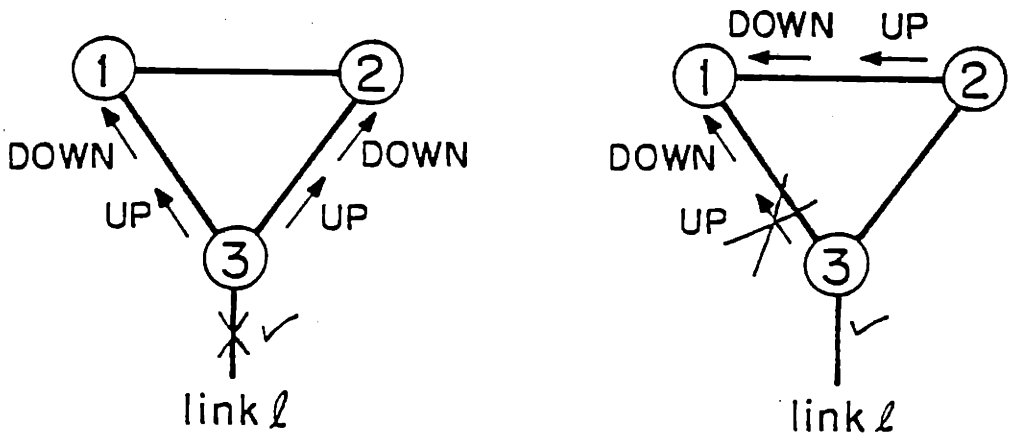

Figure 1.2.1: Topology Problem Example

messages from node 2 before receiving the first DOWN message from node 3. Now assume that before the final UP message arrives at node 1, the link connecting nodes 1 and 3 fails, and the UP message is never received. The last message that node 1 received said that link l was DOWN, yet the final state of link l is UP. Node 1 has received the topology change messages out of order. Although each link maintains order while it is operating, topology changes can result in messages being received out of order. It is quite easy to design a solution which will work for this particular example, but to find one which works for all possible cases is difficult.

The fundamental problem which the above example brought out was the difficulty in distinguishing between old and new information about the status of a link. Several obvious solutions immediately present themselves. We could tag each change message with a time stamp indicating when it was created. Or, some sequence number or counter could be used for this purpose. These are certainly reasonable suggestions, and indeed algorithms which incorporate them are used in actual networks [5, 6]. However, each of these solutions carries with it a set of additional problems. This will be discussed in more detail shortly. Practical algorithms can be devised which work properly most, or nearly all of the time. However, we seek algorithms which are theoretically guaranteed to solve the topology problem despite

arbitrary link topology changes and without placing any short term limit on the frequency of topology changes or the transmission delay of messages.

Another fundamental difficulty in solving the topology problem occurs when two sets of nodes are disconnected for a long time by one or more link failures. When these two sets (or components) are reconnected, they may have little or no knowledge of each others topology. We must find a way for both of them to adopt a single, consistent view of the network topology.

We are interested in algorithms which solve the topology problem and, in addition, have some very special properties. We desire algorithms which are event driven. This means that they transmit messages only in response to topology changes. If no topology changes occur, no messages should be sent. The reason for this is to reduce the number of messages that the algorithm sends. Algorithms which use time-outs usually do so because they are not guaranteed to work unless messages are transmitted periodically. If the time-outs are short, then many messages are sent. If they are long, the algorithm may be extremely slow in some situations. By avoiding timers altogether, we need not be concerned with this trade-off. We are interested in algorithms which are efficient in terms of the number of messages that are sent in response to probable topology changes such as a single link failure or repair. The simultaneous (or virtually simultaneous) failure of many links is such a rare event that we are not particularly concerned with the efficiency in such a case. The exception to this is a node failure in which all of the links incident on that node may fail at approximately the same time. Most nodes have only a few adjacent links, so that even when a node fails the number of topology changes is quite small.

In addition to being event driven, we seek algorithms which, whenever possible, send messages only about links which have changed status. Occasionally, this will not be possible. For example, in Figure 1.1.1 if a link between nodes 3 and 5 was suddenly repaired, node 5 would be connected to four new nodes. Node 5 may not have any information about the topology of the left hand connected set. Therefore, information about links whose status

has not changed would have to be broadcast in this case. This occurs whenever the network has been partitioned into two or more maximum connected sets which are reconnected at a later time. We refer to algorithms which try to change an existing topology in response to a topology change as delta-topology algorithms. The alternative is to rebuild the entire network topology each time a change occurs. The motivation for delta-topology algorithms is to reduce the number and size of messages that must be sent in response to most topology changes.

We also seek algorithms which do not use counters or sequence numbers to distinguish between old and new information and which do not use time-outs to decide when to transmit messages. Timers are avoided since they can be set incorrectly and must often be changed as the network grows or changes. If counters or sequence numbers are included in the algorithm messages, they must use finite length fields. These fields must either wrap around or be reset at some time. Resetting is difficult to achieve since the topology may change during the reset operation, perhaps making some nodes unreachable. When wrap around occurs we are faced with the problem of determining which of two counter values is the greater. Simply choosing a large bit field for the counter, in addition to being rather inelegant, is not a practical solution to this problem. When a partition occurs, two sets of nodes can develop arbitrarily different counter values. When the two sets are reconnected we must compare whatever counters we are using. Some of the counters may have wrapped around and we are again faced with the problem of determining which is greater. Also, after a node has failed it has no way of knowing what counter value it had been using. This does not mean to imply that algorithms which use counters cannot work. Indeed, the current ARPANET algorithm uses these methods extensively and usually works quite well. However, it is very difficult to design algorithms using these methods which are guaranteed to work properly in unusual low probability situations. For an excellent discussion on the use of counters, sequence numbers, timers, etc. in solving a generalization of the topology problem see [7].

Over the lifetime of a network, low probability events do occur, and cannot be

ignored [8]. We are interested in examining if the topology problem can be solved without resorting to counters, timers, etc. in the hope that this will help in designing algorithms which are guaranteed to work. We are also interested in understanding the trade-offs involved in this approach.

1.3 Criteria for Performance Analysis

In designing algorithms which solve the topology problem, it is useful to have some criteria by which to measure goodness. For our purposes, the most important characteristic of a topology algorithm is the worst case amount of communication that it uses in responding to a probable topology change event. We define the communication complexity, C, to be the number of messages that an algorithm sends in response to a particular event. C will usually depend on such constants as n and l, the number of nodes and links in the network. It is customary to consider the number of "elementary messages" that an algorithm sends. This is to stop an algorithm from "cheating" by sending a few large messages instead of many small ones. The algorithms that we will present typically use very short messages—usually about the same number of bits as the DLC will add on for framing and error control. We are not very concerned about taking into account the size of messages, and are perfectly content to consider C to be the actual number of messages sent. In addition to the actual value of C, we are concerned with its order, that is, how it grows with increasing n or l. When we write $C \sim O(n)$ this means that there is a constant k such that $C < nk$ [9].

Another traditional measure of the goodness of a distributed algorithm is its time complexity, T. For a given topology change event, T is the maximum number of time units until the algorithm terminates, if the transmission of each message is assumed to take at most one time unit. As with C, we are mainly concerned with the value that T takes on for probable topology changes such as a single link failure. We can also define the order of T in a similar manner as was done for C. The important time for topology algorithms is how long it takes for each node in a maximum connected set to become aware that a link has failed. Depending on the particular algorithm, this time may be much smaller than

T. However, T does give an upper bound on this time. We are mainly concerned with the order of T. For example, we would not want T to grow exponentially with the size of the network for a common topology change.

We mentioned earlier that the topology problem should be solved after no status changes have occurred for a sufficient but finite time. T determines the required time for a particular network topology. The algorithms that we will present can be expected to take anywhere from a fraction of a second to a few seconds to run. However, the average time between topology changes in the types of networks that we are concerned with is at least hours, and is probably days, weeks or months. Thus, having a good estimate of T is not particularly important, provided that it is much less than the mean time between failures.

1.4 The Routing Information Problem

Many types of routing algorithms require some measure of the delay or cost of sending a message packet in each direction, on each link in the network, in order to make their routing decisions [1, 4]. We refer to this as requiring global information. Each node periodically or continuously measures the delay in transmitting packets on each of its adjacent links. We are not concerned with how or how often these measurements are made. They can be absolute, averaged, scaled, etc.. We refer to these measurements as "delay measurements" or more generally as "routing information." The actual values are not our concern as long as they have some significance to the particular routing algorithm being used, and can be adequately represented using a finite length field. At any time, a node may produce a new value for the routing delay on one of its outgoing links. The node would like to broadcast the new value to each node to which it is connected and be sure that each connected node receives the new delay measurement even when topology changes occur. We call this the routing information problem. Any routing algorithm which requires global information must solve this problem in some way.

It is apparent that the routing information problem is quite similar to the topology problem in many respects. We can consider it to be a generalization of the topology problem

which allows directed links to have any one of a set of possible delay values. One such value would correspond to infinite delay meaning that the link has failed. However, there are also some important differences between the two problems. A change in the value of a delay measurement has a much lower priority then a topology change. Typically such a delay change will result in a slight change in the current routing being used by the network [10]. We are not overly concerned that this delay change be broadcast as quickly as possible. A delay change is a much more common event than a topology change. A normally functioning network can be expected to generate delay changes as an ordinary consequence of changing traffic patterns. Since delay changes are quite common, we are even more concerned with minimizing communication than we were with topology changes.

The properties that we desire in routing information algorithms are the same as those desired in a topology algorithm. We want event driven algorithms which transmit messages only when delay values change. We would also like to send delay information about only those links whose delay value has changed, whenever possible. Timers, time-outs, counters and sequence numbers will not be used for the same reasons mentioned earlier. The most important characteristics of the routing information algorithms that we will consider are that they are guaranteed to work in the presence of arbitrary topology changes, and that they minimize communication during normal operation. Of course, we also want routing information algorithms which are independent of the particular routing algorithm used.

In practical networks [6], and in the literature [11], the topology problem and the routing information problem are usually solved as an integral part of the routing algorithm. This usually involves significantly complicating an already complex routing algorithm and develops a solution which is highly dependent on the particular routing method used. In this thesis we are interested in studying the topology and routing information problems separately from the actual routing algorithm. In this way we can develop solutions which are more widely applicable. By dividing the overall problem, we hope to make it easier to solve. For example, once the topology problem has been solved, we may be able to use

this solution to simplify the routing information problem. Solving both of these problems greatly simplifies the design of the actual routing algorithm since all of the information that is needed to make routing decisions is available and reliable at each node.

CHAPTER 2

Summary and Analysis of Previous Work

In this chapter we present and analyze one algorithm for solving the combined routing information and topology problem, and two algorithms for solving the topology problem alone. Each method accomplishes slightly different tasks and so direct comparison is difficult. The last two algorithms are event driven and can therefore be analyzed in terms of their time and communication complexity. We present this analysis to serve as a basis for comparison to the algorithms which will be discussed in Chapter 3.

2.1 The ARPANET Update Policy

The current algorithm used on the ARPANET solves both the topology and routing information problems together, and is known as the Update policy of the routing algorithm. The algorithm is not, strictly speaking, event driven. It does transmit messages in response to topology or delay changes, but it also requires that this information be transmitted at least once a minute even if it has not changed. The algorithm uses a combination of time-outs sequence numbers and packet aging to insure reliability in nearly all situations. It is a carefully designed practical solution to the problems that we have discussed. The following is a brief summary of some of the important properties of this protocol [5].

Each node measures delay on each of its outgoing links over a 10 second interval. If the measured delay value differs from the previously transmitted value by more than a certain threshold, a new message called an update packet is formed which contains the new measurement. The threshold is a monotonically decreasing function of time which eventually becomes zero, so that after a certain time an update packet will be created even if the delay on a link has not changed. A failed link is considered to have a very large delay and is handled immediately.

Each update packet produced at a node is labeled with the identity of the node, a sequence number, and an age field. Update packets are then broadcast to every connected node using "flooding." In flooding, each node broadcasts a received packet on all its links including the one on which the packet was received. This echo message serves as an acknowledgement. If it is not received within a time-out period, the packet is retransmitted. When a node receives an update packet, it always sends an acknowledgement echo to the transmitting node, but must decide whether to accept this new packet based on the sequence number and age fields. Each node keeps the sequence number and age for the last packet that it accepted from a given node. The age fields are all incremented with each clock tick until they reach a maximum value. When a node receives an update packet from some other node it checks the age field of the last packet received from that node. If the age field is at its maximum then the old update is "too old," and the new one is accepted and forwarded to all neighbors. Otherwise, the sequence numbers of the old packet and the one just received are compared (modulo 2). The new packet is only accepted if it has a higher sequence number than the previously accepted packet. The first packet received from a given node is always accepted.

When a node fails, it must be prohibited from restarting until all of the update packets which it had previously sent have reached maximum age. Then, any new messages that it transmits will be accepted. After a node becomes operational, it will receive an update packet from each connected node, since they must be transmitted at least once in a given time interval. Thus a node will automatically receive the latest delay measurements a finite time after being reconnected to a set of nodes.

The ARPANET procedure works well but has some disadvantages. The most important is that there are certain very low probability events for which the algorithm will not function properly [8]. In addition, it involves proper settings of several different parameters: the maximum age of packets, the minimum interval between creation of update packets, the maximum interval between creation of update packets, etc. . Trying to determine the optimum settings for each of these parameters can result in contradictory goals. Also, as

the size of the network grows some of these parameters need to be changed; they are not self-adjusting. In addition, it is theoretically possible for packets to avoid aging and to circulate indefinitely. Perlman [7] has suggested several modifications to the update policy which improve performance and reduce dependency on timing parameters.

Since the ARPANET update procedure is not purely event driven it cannot be said to have a true time or communication complexity. However, we can still analyze its behavior in some simple situations. When a single link changes delay above a threshold, it results in a flood operation which sends one packet in each direction on each link. Therefore for this case, $C = 2l$. If each of these packets takes at most one time unit then the last message must be received within $d + 1$ time units, where d is the diameter of the network in hops. The $+1$ accounts for the final echo acknowledgement. The diameter of an n node network is upper bounded by $n - 1$ therefore in this case $T = n$. In summary, for the ARPANET update procedure responding to a delay change on a single link we have:

$$C = 2l \sim O(l) \qquad\qquad T = n \sim O(n)$$

2.2 The Finn Algorithm

An algorithm which can be adapted to solve the topology problem was presented by Finn in 1979 [12]. The algorithm, as it was originally presented, does not actually result in each node knowing the network topology. Rather, each node determines which other nodes it is connected to. In addition, the Finn algorithm has a special property called resynchronization which will be described shortly. Roskind [13] modified the Finn algorithm so that it would also compute the network topology. The modification does not alter the structure or logic of the algorithm, but does require that additional information be stored at each node and be transmitted in each algorithm message. We proceed to analyze some of the characteristics of this algorithm. What we will refer to as the Finn algorithm is actually Roskind's modification.

We are interested in the Finn algorithm because it has several of the properties which we have called desirable. The algorithm is purely event driven. It sends messages

only when link topology changes occur. The algorithm is also free from timers and time-outs. It uses counters while it is running, but after termination old counter values are not needed. In addition, the maximum value of these counters is strictly bounded and wrap-around presents no problems. Unfortunately, the algorithm is not of the delta-topology type. It responds to each topology change, no matter how minor, by ignoring the old network topology, and completely reconstructing a new topology. When a single link fails or is repaired the entire network topology is rebuilt. The algorithm unquestionably works properly in any topology change situation, but is not at all efficient in communication. This is not surprising since efficiency was not one of Dr. Finn's goals in designing the algorithm. He was interested in proving that such as algorithm could be used to provide a perfectly reliable end to end packet delivery protocol. Such protocols are outside of the scope of this thesis.

In addition to solving the topology problem the Finn algorithm has a set of properties collectively called resynchronization. For the purposes of this thesis we will consider an algorithm to accomplish resynchronization if it has the following properties:

1) All nodes in a connected set must start running the algorithm before any node stops running the algorithm.
2) When a node stops running the algorithm, it sends a message to each of its neighbors. Upon receipt of this message, each neighbor will either stop running the algorithm or will have started running a new version of the algorithm.
3) Before the algorithm can terminate at any node, at least one message must have been sent in each direction on each link.

It is not at all clear, at this point, why we should want a topology algorithm to have these particular properties. Indeed, if all we are interested in is solving the topology problem, there is little motivation to consider resynchronization. In Chapter 4 we shall consider the routing information problem. We will show that the solution to this problem is greatly simplified if a topology algorithm is available which has the resynchronization properties. Until then, we shall accept resynchronization as a desirable property of topology algorithms when solving the routing information problem is also being considered.

We will present a very simplified picture of the Finn algorithm. Each node is either in normal mode (not running the algorithm) or resync. mode (running the algorithm). When a node detects a status change, it enters resync. mode and sends a message to each of its neighbors. The message consists of a resync. count, a node table N, and an edge table E. These tables are also stored at each node. We will not be concerned with the resync. count. The node table N, at some node n, contains an entry for each node in the network. Each entry N_i can take on one of three possible values.

$N_i = 0$: Node i has not been heard from since the beginning of the algorithm.

$N_i = 1$: A message has been received from node i.

$N_i = 2$: Node i has received an algorithm message on each of its working links.

The E table stored at each node contains an entry for each link in the network. Each entry E_l can take on one of two possible values.

$E_l = 0$: Link l is not operating.

$E_l = 1$: Link l is operating.

When a node n starts the algorithm, it knows that it is connected to itself, and sets $N_n = 1$. The rest of the entries in N and E start out at zero. When a node receives a message over a link l, it knows that link l is working and can set $E_l = 1$. When node n has received a message over each of its working links, it sets $N_n = 2$. Other than the above internal changes to N and E, a node only modifies these tables as a result of the information contained in an algorithm message. A received message contains a neighboring node's N and E tables. The node combines its own tables with those received by taking an entry by entry maximum. This has the effect of a node adding a neighbor's knowledge of the network topology to its own. The node then stores these combined tables and also transmits them to each neighbor. Each time a node receives a message, it modifies its E and N tables accordingly, and then transmits these tables to each neighbor. Thus, any time a node gains further knowledge of the network topology it informs each neighbor. In this way, complete knowledge of the network topology is gradually gained by each node.

Eventually, each node n makes its own entry in its node table $N_n = 2$, and sends

this to each neighbor. When a node's N table contains entries which are all 0 or 2, it has heard from every connected node and can terminate the algorithm, after transmitting its tables. This can be seen as follows. When a node n makes its own value $N_n = 2$, it must have heard from each of its neighbors, and must have $N_i \geq 1$ $\forall$ neighbors i. Therefore, the message that it sends when it attains a value of $N_n = 2$ must also contain $N_i \geq 1$ for each of its neighbors, i. Any time a node has $N_n = 2$ for some node n, it must also have $N_i \geq 1$ for each of node n's neighbors, i. Therefore, if a node has all entries in its N table equal to 0 or 2 there cannot be any other connected nodes which it has not heard from. It can be shown that when the termination condition is reached, the topology table E is also complete and correct.

The algorithm also contains a mechanism for detecting further changes in the topology while it is running. When this occurs, each node begins building the topology over again.

We are interested in finding the time and communication complexity of the Finn algorithm. As the algorithm is presented by Finn and Roskind, it can require unbounded communication. This is due to a node transmitting messages each time it receives a message. If we make no assumptions about transmission delays, it is easy to see that a group of nodes will keep exchanging messages while they wait for some other message to arrive over a slow path. If there is no limit on the transmission delay on the slow path, there can be no bound on the number of messages sent. This problem can be solved by adopting the simple rule that nodes only send messages when their N tables change. It is clear that the algorithm must still work since we are only eliminating duplicate messages.

We consider first the case of a single link topology change, and try to find an upper bound on the number of messages sent. Let a network have l links and n connected nodes, after the topology change. We are only concerned with these connected nodes. We will establish a bound on the maximum number of times that a node's N table can change. Each entry N_i can go from 0 to 1, and then from 1 to 2. This means that the table can

change at most $2n$ times. However, except for the node which started the algorithm, each node begins with at least two N_i's = 1 (itself, and the first node that it received a message from). Therefore, a node's N table can change at most $2n-2$ times. This means that each node sends at most $2n-2$ messages on each of its links. We define a port to be an end of a link. Thus there are $2l$ ports in the network. Each node sends $(2n-2)$ messages on each of its ports. Therefore, a total of $2l(2n-2)$ messages are sent on the network. The node which starts the algorithm also sends one additional message on each of its ports. This message has at only one $N_i = 1$ (itself). Since the starting node can have at most $n-1$ ports, the communication complexity for a single link topology change is bounded by:

$$C \leq 2l(2n-2) + (n-1)$$

or:

$$C \leq (4l+1)(n-1)$$

$$C \sim O(nl)$$

It can be shown that this bound is satisfied with equality for a fully connected network with appropriate message timing. In addition to the large number of messages that the algorithm sends, each individual message is quite large. The messages in the Finn algorithm contain information about each node in the network. If message size were incorporated into C, it would become even larger.

We can obtain a best case bound by noticing that there is a constraint on how rapidly (in terms of the number of changes) a node's N table can change. Define k to be the number of entries in a node's N table for which $N_i = 2$. At the start of the algorithm, $k = 0$. When a node terminates the algorithm, $k = n$. Each time k changes a node must transmit a message to each neighbor since the node's N table must have changed. It can be shown that a node's value of k can increase by at most 2 in response to receiving a single algorithm message. A node's value of k must go from 0 to n and can increase by at most 2 each time it changes. Therefore, a node's value of k must change at least $\frac{n}{2}$ times. Each node sends at least $\frac{n}{2}$ messages on each of its links. Therefore, the total number of

messages sent is lower bounded by $2l\frac{n}{2}$. This lower bound is not at all tight; the Finn algorithm usually sends at least twice as many messages as this bound requires.

In summary, the bounds on C for the Finn algorithm are:

$$ln < C \leq (4l+1)(n-1)$$

$$C \sim O(ln)$$

For an arbitrary number of topology changes, the worst case is for the algorithm to be nearly finished with one change when the next topology change occurs. Therefore, for k topology changes we have:

$$C < k(4l+1)(n-1)$$

The time complexity of the Finn Algorithm for a single topology change can be bounded very simply. Let d be the diameter of the network. After d time units, each node has started the algorithm. After one more time unit, each node n has $N_n = 2$. After d more time units, each node has $N_i = 2$ for each connected node i, and after one more time unit the algorithm must have terminated at each node. The diameter of the network is upper bounded by $n-1$, so we have the following bound on the time complexity.

$$T < 2n \qquad\qquad T \sim O(n)$$

2.3 The kSTRA Algorithm

Roskind [13] has developed a topology algorithm for dealing with a few special cases of the topology problem which is very efficient in communication. This algorithm, called kSTRA (k Spanning Tree Resynchronization Algorithm), works properly for a small number of link topology changes provided that there are no node failures or component reconnects. In all other cases, the Finn algorithm must be run. The kSTRA algorithm is not self sufficient, and must work in conjunction with the Finn algorithm. kSTRA is event driven and is delta-topology, but it uses a counter which must be stored between successive runs. This counter is strictly bounded (in fact, it is binary), but it presents difficulties if

we try to generalize kSTRA to deal with component reconnects and node failures. We are interested in kSTRA because its structure is similar to one of the algorithms which will be presented in the next chapter. Indeed, kSTRA served as the major inspiration for this work.

After the Finn algorithm terminates, each node in a connected set is guaranteed to have the same topology table, E. kSTRA uses this fact to generate the same set of edge disjoint spanning trees at each node. The number of edge disjoint spanning trees that exist in a network is limited by the degree of the lowest degree node in the graph which describes the network topology. There are typically only one or two edge disjoint trees in a network. The idea of kSTRA is to use these precomputed spanning trees to communicate topology change information. For a node k to send a message to all other nodes in the network requires at least $n-1$ messages. If k used a spanning tree to send this message, it would require exactly $n-1$ messages. Therefore, the use of a spanning tree to communicate messages is optimal in this sense. kSTRA reduces the number of messages which must be sent by communicating most of its messages over spanning trees.

The algorithm has three phases. The first is the notify phase. When a node k detects a link status change, it finds an intact spanning tree (one which has not been damaged by the status change) and uses the tree to notify each connected node of the change. If no intact trees remain, the Finn algorithm is run. We call this process an update. When the leaves of the tree are notified of the topology change, they send acknowledgements along the tree which collect at node k. This is called the acknowledgement phase. When node k receives acknowledgements on each of its adjacent spanning tree links, it is sure that each connected node has acknowledged the update. Node k then starts the third phase, called termination. It floods a message to all connected nodes, telling them that the update is over and that they can terminate the algorithm. A flood is used in the final phase to provide the same type of termination which occurs in the Finn algorithm.

kSTRA also contains counters and version numbers to deal with topology changes which occur while the algorithm is running. When this occurs, the various update messages

eventually collect at a single node where a larger update is formed containing all of the changes. This large update is then dealt with as before.

kSTRA provides a type of resynchronization, but does not have all of the properties that we mentioned in the previous section. Specifically, kSTRA does not require that at least one message is sent in each direction on each link before the algorithm can terminate. While this is not important for solving the topology problem, it will be significant when we consider the routing information problem in Chapter 4.

It is very simple to analyze the time and communication complexity of kSTRA for single link topology changes. The algorithm sends two messages on each spanning tree link during the notify and acknowledgement phases, and two messages on each network link during the final flood. Therefore, the communication complexity is:

$$C = 2(n-1) + 2l \qquad\qquad C \sim O(l)$$

The time complexity is bounded by three times the diameter since each phase must propagate across the network. For simplicity we write:

$$T < 3n \qquad\qquad T \sim O(n)$$

Since kSTRA does not work for general topology changes, it is pointless to analyze the time and communication complexity for multiple changes.

CHAPTER 3

Algorithms for Broadcasting Topology Changes

Each of the three topology algorithms which were discussed in Chapter 2 had its own advantages and disadvantages. The ARPANET update procedure has worked well for several years, but can fail in some situations, and it is not event driven. The Finn algorithm always works, but it is not efficient in communication. Roskind's kSTRA algorithm is very efficient in communication, but only works in a few special cases. In this chapter we present two algorithms for solving the topology problem which attempt to combine the efficiency of kSTRA with the generality of the Finn algorithm. The algorithms that we present will be free from time-outs, counters, sequence numbers or timers, and will be event driven and delta topology. The first algorithm, called the Flooding Topology Algorithm (FTA), has the resynchronization property of the Finn algorithm. This will be used in Chapter 4 to provide a simple solution to the routing information problem. FTA provides resynchronization and efficiency at a price: it is considerably more complicated than any algorithm which we have so far discussed. The second algorithm which we will discuss is called the Shortest Path Topology Algorithm (SPTA). It does not have the resynchronization property, but it is extraordinarily efficient for simple topology changes. In Chapter 4, SPTA will be extended to deal with the routing information problem.

3.1 Topology Algorithm Assumptions

Before presenting the algorithms, we must have a clear understanding of the exact conditions under which they will operate. Both algorithms make the same assumptions about the behavior of the network links and nodes, and these assumptions are critical to their operation. Several times we have referred to a link as being either operating or not operating. Even this simple idea needs some clarification since at any time the two end nodes of a link are making independent judgements about the link's status.

At each node of a link there is a data link controller (DLC) which governs the transmission and reception of data over the link, and decides whether or not the link should be used. Let l be a link joining nodes n and m. The DLC's at n and m have an established protocol for the exchange of packets. If errors occur in a packet it is retransmitted, perhaps several times, until it is correctly received. The DLC can then release the packet for processing at the node. To the outside world, the DLC's provide an error free communication path. In addition, the path preserves order in the following sense. If we consider all packets traveling from n to m over l to have sequence numbers, then if a packet arrives at m and is released by m's DLC, it must have a higher sequence number than any packet previously released by m's DLC.

If the error rate on a link becomes intolerably high (based on some criteria) the DLC at a node may decide that the link is unusable. The exact criterion used is arbitrary. It is also possible that the link has actually broken, in which case no information is received on it. At any time, each DLC either considers a link to be operating and calls it up or not operating and calls it down. This information is available to a topology algorithm running at a node. We assume that a node does not receive messages over a link while its DLC considers the link to be down.

We require that DLC's obey certain rules in declaring links to be up or down. If a node n determines that node m's DLC has declared link l to be down, then it must also declare it to be down. Furthermore, before a link can be put back into operation, the DLC's at each end node must agree that the link is operating. These requirements imply the following rule: a link must be called down by both end nodes before it can be called up by either. Figure 3.1.1 illustrates the ordinary sequence of status changes when a link fails and is repaired at a later time. Notice that during the intervals τ and τ' the two nodes have inconsistent views of link l's status. No assumptions are made about the size of these intervals other than that they be finite; however for efficient operation of the topology algorithms that we present, these intervals should not be much longer than the ordinary transit time of a packet.

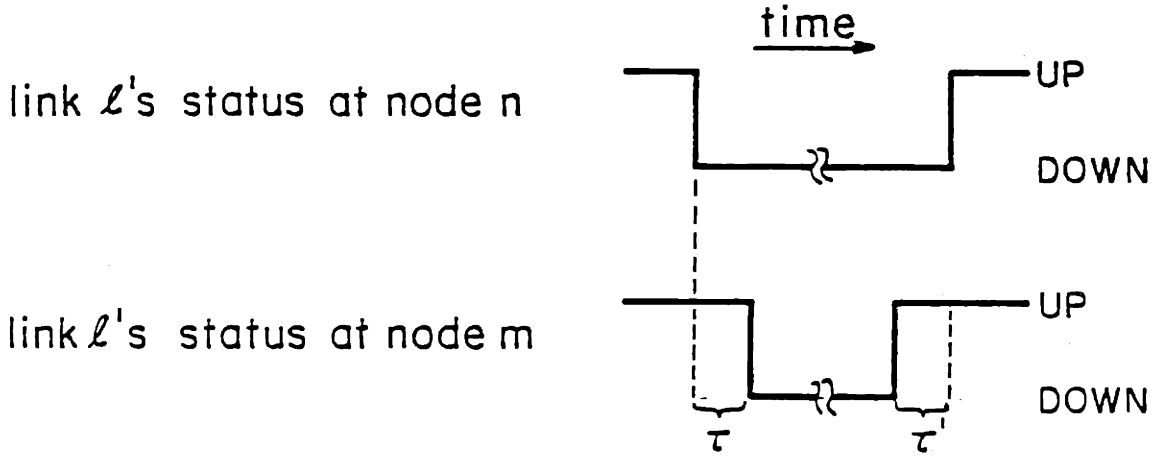

Figure 3.1.1: Ordinary DLC Status Change

Figure 3.1.2. illustrates an unusual but possible sequence of changes. Nodes n and m decide that link l should be put back into operation. Node n makes the status change, but m (perhaps because the link fails, or a high error rate is suddenly detected) changes its mind and leaves the link's status as down. A short time later, n realizes that m is keeping link l down and must also declare it to be down. The result is that two status changes occur at one end node, and none occur at the other end node. This event will not occur very often, perhaps almost never, but our algorithms must be designed to work properly if it does.

In the following algorithms, when we refer to a link as "operating," we mean that both of its end nodes consider it to be up. When a link is called "not operating," at least one of its end nodes considers it down.

3.2 Flooding Topology Algorithm (FTA)

Before presenting FTA, we will briefly review the motivation that leads to its structure. We desire a topology algorithm which is event driven. In addition, we would like the algorithm messages to contain information only about links whose operational status has changed, whenever possible. Since information about a link may be generated each time its

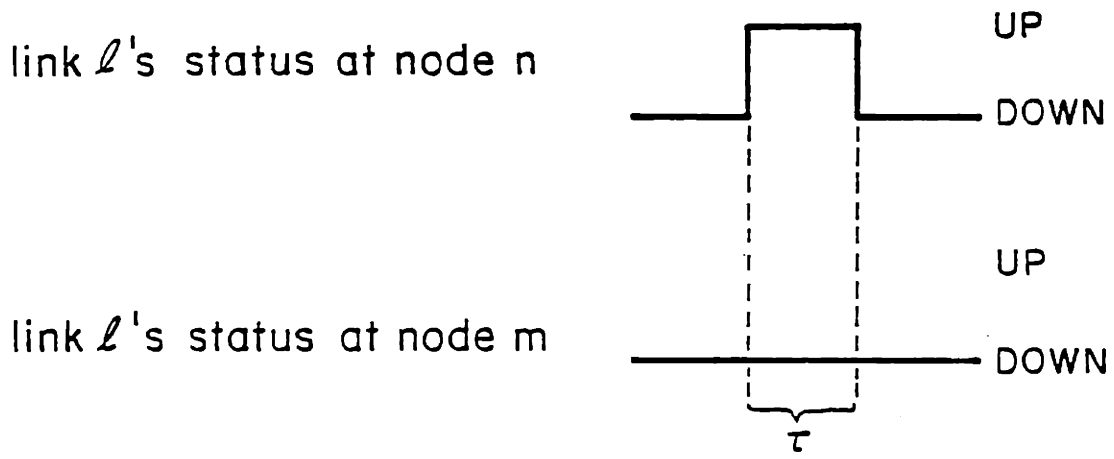

Figure 3.1.2: Unusual DLC Status Change

status changes, we need some way of either distinguishing old, outdated information from new, current information, or a way of restricting the entry of new information into the network. The most obvious way of accomplishing this is to use counters on the status update information about a link, and then have a node only accept updates with a higher count then it has previously seen. We wish to avoid counters because of the problems that they pose when trying to reconnect two sets of nodes which have been disconnected for a period of time. Since the counter must in practice be bounded, it could have wrapped around in one set and not in the other. Also, if a node fails it may loose track of any counter which it had been using.

This algorithm uses acknowledgements to limit the new information that can be introduced about a given link. When each node has acknowledged the receipt of old information, then new information can be sent. This scheme eliminates confusion between old and new information, but requires we insure that acknowledgements eventually get back to, and collect at a node. This is not easy since as acknowledgements travel back to the source, any path being used may break.

In FTA, when several links change status, we combine this information into one

large topology update message. By combining messages in this way, we help to ensure that an acknowledgement about each change will be received. A node can then acknowledge this combined update with a single message. Unfortunately, combining updates in this way also creates a problem. A node which started an update about a given link may not be the node to which acknowledgements will eventually collect, since its update may be combined with one started by a different node. Thus a node is never quite sure when it can introduce a new update about one of its adjacent links, since it doesn't know when all of its neighbors have acknowledged the old change. This problem can be solved by having the node at which all acknowledgements collect broadcast a message of the form "the update is over" on every working link in the network. Thus a node knows that once it receives and sends this final message, all of its neighbors must also receive this final message before they receive any new information that the node sends.

This leads to an algorithm with three phases.

1) Broadcast updates to every node.
2) Receive acknowledgements from every node.
3) Broadcast "update over" message on every link.

This may sound like a large amount of information to be sent about a single topology change, and indeed it is. In this sense the method is not very efficient in communication, but it does allow us to send information about only links that change (most of the time), and only when they change. This is where the real communications resource savings come in.

FTA is conceptually a very simple algorithm, but a precise statement of the procedures to be followed at each node involves handling many special cases and is somewhat complicated. We begin by examining the data structures used in the algorithm. Each node n maintains a table T which contains an entry $T(j)$ for each link j in the network. $T(j)$ is the operational status for link j that FTA reports to the outside world; it does not necessarily equal $s(j)$, the actual DLC reported status of link j at node n. This is caused by the rules that the algorithm establishes for allowing new information about a link to be

reported, and will be discussed shortly.

We refer to an "update" as the process of making one or more changes to the network topology under the direction of a controlling node c. The messages used by the algorithm contain an update table U which lists the changes being made to the topology, and the number of the controlling node that is directing this set of changes. The update table U and the controlling node c uniquely define a specific "version" of the algorithm. We also use the term update to refer to an algorithm message containing an update table and a controlling node.

Each entry $U(j)$ in the U table tells a node what to do with the status of a link j. $U(j)$ can be one of the following: UP, DOWN, or NO_CHANGE. We say that the topology at a node is changed when the information contained in the U table is used to modify the T table. If $U(j) =$ DOWN and $T(j)$ is already equal to DOWN, we still refer to setting $U(j) := T(J)$ as installing a change in the topology.

Whenever a node has a U table which contains some $U(j) \neq$ NO_CHANGE we say that FTA is running at a node, otherwise it is said to be terminated. The algorithm is said to have terminated when it has terminated at each node in the network, and no algorithm messages are present on any link.

It was mentioned earlier that we establish rules to limit the introduction of new information about the status of a link. A node introduces new information by combining its current DLC status of a given link j with its U table. We establish the following rule (or precedence) in the modification of an entry $U(j)$ in a node's U table. If the values that $U(j)$ can take on satisfy: DOWN > UP > NO_CHANGE, then a node may never lower the value of $U(j)$ except when terminating the algorithm. This can be restated as follows. A node may not start an update which will turn a given link up if there is a current update which is attempting to turn the link down. Thus down takes precedence over up.

We mentioned earlier that when a node becomes aware of two different updates, it combines them. We now discuss the specific rules used to accomplish this. Let a node

with an update table U receive a message containing an update table U'. The first step is to take the maximum entry contained in the tables for each link j. This enforces the above precedence rule. We also adopt the following rule. If a link is being turned down, we do not allow any other links listed as down in node n's topology table to be turned up. A node uses this same rule when it detects an up status change on one of its adjacent links. Essentially this means that the algorithm first handles down changes, and after these are completed it handles any up changes. This rule is used to simplify the reconnection of disconnected components. This will explained in more detail shortly.

If a node becomes aware of two update tables which differ only in the controlling node, it ignores the one with the lower controlling node number. Thus, when the U tables of two updates are the same, the update with the higher controlling node number takes precedence.

Having described what updates are and how they are combined, we can explain how the algorithm modifies topologies according to the information contained in updates. The algorithm is a set of rules for the transmission of messages and the modification of a node's topology table. A node produces messages in response to detecting a status change in an adjacent link or in response to a received message. Before describing the exact procedures to be followed at a node, we will examine how the algorithm behaves in response to a simple failure event.

Consider the network shown in Figure 3.2.1 to be in steady state at time t. By this we mean that each network node has an identical topology table T, and that the algorithm is not running. At some later time t_1 link A either physically breaks, or experiences an intolerably high error rate. This is detected, at different times, by nodes 2 and 5. Nodes 2 and 5 each start a version of the algorithm with $U(A) = \text{DOWN}$, and itself as the controlling node. The two versions which are started have the same U table, but different controlling nodes. Nodes 2 and 5 each send update messages of the form $\text{UPDATE}(U, c)$ on each of their operating adjacent links. When a neighboring node, such as node 1, receives an update

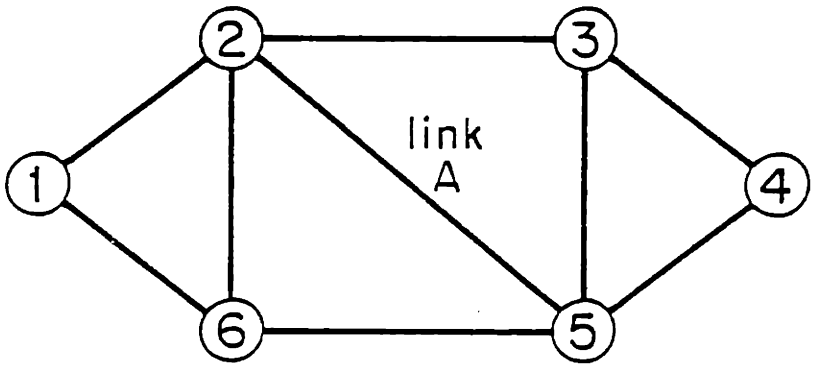

Figure 3.2.1: A Simple Network

message, it recognizes that this is a new update (in the sense that its own U table did not previously have this information), and it propagates the update by repeating the message on each of its operating links, except for the one on which it first received the update message. The link on which a node first receives an update message for a particular algorithm version is called the respond link, r, for that version. Each node running a version of the algorithm has a respond link except for the controlling node for that version.

Consider for the moment only the version of the algorithm started by node 5. Node 5 sends update messages to nodes 3, 4, and 6. Nodes 3, 4, and 6 each repeat this message on each of their adjacent links, except for the link connecting them to node 5 (their respond link). The update message propagates out to every node in the network in this fashion. Eventually, for this simple failure event, a node which has sent an update message on a link will receive on that link the exact same message (same U table and controlling node) which it has sent. This echo serves as an acknowledgement. When a node has received acknowledgements in this fashion on each of its operating links, except for its respond link, it can then send an acknowledgement on its respond link. The acknowledgement that it sends is again merely an echo of the update message that it originally received. Thus the only thing that distinguishes an acknowledgement message from an update message is how

it is interpreted by the receiving node.

Eventually, the controlling node, in this case node 5, will receive an acknowledgement on each of its operating links, since it has no respond link. When this occurs, node 5 is sure that each network node to which it is connected has been made aware of this update. Node 5 then sends a special UPDATE_OVER(U, c) message to each of its neighbors. This message propagates out to each network node in the same way that an update message does, except that no acknowledgements are needed. In our example, when the UPDATE_OVER message has reached every node connected to the controlling node, the algorithm terminates.

The propagate – acknowledgement phases can be understood more clearly by applying some of the graph theory used in Chapter 1. The propagate phase is really constructing a directed, rooted spanning tree with the controlling node as the root. The spanning tree is the collection of all the respond links in the network. Acknowledgements then travel from the leaves of this tree to the root, using the respond links. When the controlling node receives these acknowledgements, it is sure that every node in the spanning tree has acknowledged the change. The propagate – acknowledgement phases together transmit exactly one message on each directed link. The terminate phase cannot begin until each of these messages is received. This is how FTA meets the resynchronization requirement discussed earlier.

We mentioned earlier that node 2 had also started a version of the algorithm. What has happened to it? After a node has received and propagated the version started by node 5, it will ignore any messages from the version started by node 2. This is due to our rule for combining updates which says: if the U tables are identical, propagate the version with the higher controlling node number. Thus the version started by node 2 will quietly "die," while the version started by node 5 will take over.

Since the version started by node 2 was never completed, any messages which it produced were wasteful. It is interesting to consider whether these messages could have been

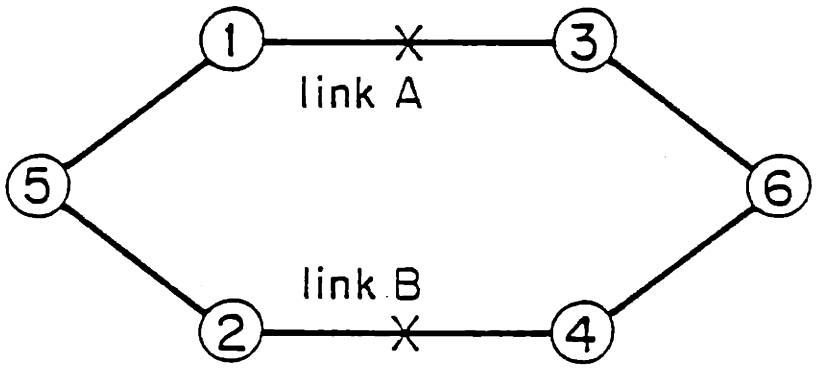

Figure 3.2.2: Multiple Failure Example

avoided. When node 2 detects that link A is down, it knows that node 5 will eventually detect this as well. Node 2 also knows that it has a path to node 5 which does not use link A (for example 2–3–5). Therefore, if node 2 did not start a version of the algorithm, node 5 still would, and each node would still correctly implement the change. This suggests the following rule for a node to start an update. Let link A connect nodes n and m where $n < m$. If link A fails, node n should not start an update if there is a path between n and m which does not use link A. Unfortunately, this rule does NOT work properly. In Figure 3.2.2, assume that links A and B fail at approximately the same time. If the above rule is applied, neither node 1 nor node 2 will start an update. Node 5 will never find out about either of the two changes. Therefore, each node adjacent to a failed link must start an update.

This point was mentioned for two reasons. The first was to illustrate an important property of topology algorithms: perfectly reasonable rules often don't work properly when multiple failures occur. The second thing to be learned from this example is that there is a fundamental difference between turning a link up and turning it down. When a link is being turned up, both end nodes can communicate with each other and decide on the best action to take for minimizing the number of messages sent. For example, in FTA, when a

link is being turned up only the end node with the higher node number will start an update. When a link is turned down, no such arrangement can be made, and both end nodes must start an update.

When multiple link status changes occur, several versions of the algorithm are started, at least one at the site of each change. Eventually, at least one node will receive two different updates and will combine them as described earlier. This combined update represents a new version of the algorithm, which is then propagated. Updates continue to be combined in this manner until at least one node becomes aware of each change which can be introduced according to the combination rules. Exactly one version of the algorithm (the one which contains each change and the highest controlling node number) will continue on to completion, and the rest will die out. The versions which die out represent unwanted, wasteful communication, but their presence is required, within this type of algorithm structure, to ensure that the algorithm works properly in the presense of an arbitrary set of topological changes.

There are times when an up change about a link l will be suppressed for a time. This can happen in two ways.

1) The up status change can occur while each of l's end nodes are turning a link down.

2) A down change about some link can occur before link l's up change completes.

In the first case, l's up change is delayed until any down changes are completed. Then, a new version of FTA is started to bring l up. In the second case, an update is started to turn l down, and after this completes, another update is started to bring l up. Either way, an update containing the correct status of l is eventually started.

Occasionally, when a link is turned up this will result in the reconnection of two previously disconnected components. Since these two sets of nodes may have arbitrarily different topologies, we need a way of having each node involved adopt a consistent, correct topology. We adopt the following two rules.

1) A node records as down each link to which it is not connected according to its topology table T.

2) When a node detects a reconnect of two components, it includes in the update list an up change for each link l for which $T(l) = \text{UP}$.

Therefore, when a reconnect occurs, the update list eventually contains an up change for each operating link in a maximum connected set. When this update list is installed at any two nodes in the set, they arrive at the same topology table. In this way, a consistent topology is adopted throughout the set of nodes.

It is possible for a reconnect to be started and then to have some link fail. By applying the update combination rules mentioned earlier, we can see that the reconnect is stopped, and the link which was joining two components is turned down. However, the update list also contains, by the above rules, an up change for every working link in either component. We can now see how the rule for disallowing simultaneous up and down changes is applied. That rule states that when a down change occurs we do not allow any links listed as down in a node's topology to be turned up. In the case of a component, the links listed as down include all links in the neighboring component. This is true because a node always considers disconnected links to be down. All of the up changes for links in adjacent components are converted to down changes. However, an up change for a link inside a given component is allowed to continue if each node in the component already lists the link as up in its topology table. Thus, the up changes which remain have no effect on the topology. These changes are left in the update list to enforce the rule that each $U(j)$ can not decrease. This property of $U(j)$ is very useful in showing that FTA works correctly.

FTA Algorithm Data Structures: contained at node n

$s(j)$ = the current DLC status of each adjacent link j
{UP, DOWN}

$T(j)$ = node n's current topology information for each link j
{UP, DOWN}

$U(j)$ = a change in the status of link j
{UP, DOWN, NO_CHANGE}

$L(j)$ = whether n is waiting for an acknowledgement on link j

{WAITING, NOT_WAITING}

c = the node controlling the current version of the algorithm

R = the adjacent respond link on which an acknowledgement will be sent

Notes:

1) A primed variable refers to one received from a neighboring node.

2) U or T by itself refers to a table of values for each link.

3) DOWN > UP > NO_CHANGE

The functions *node_connect* and *link_connect* defined below are useful in detecting disconnected links.

$$node\_connect(i,k,T) = \begin{cases} \text{YES} & \text{if nodes } i \text{ and } k \text{ are connected according to topology } T. \\ \text{NO} & \text{otherwise.} \end{cases}$$

$$link\_connect(i,j,T) = \begin{cases} \text{YES} & \text{if node } i \text{ is connected to link } j \text{ according to topology } T \\ \text{NO} & \text{otherwise.} \end{cases}$$

We have already discussed the two principal messages that the algorithm uses. However, there are two more which must be explained. We mentioned that when two end nodes decide to bring a link up we would like only the one with the higher node number to start an update. FTA accomplishes this synchronization by using a message called BRING_LINK_UP. Exactly how it is used will be discussed shortly.

The message BRING_LINK_DOWN is used when it is necessary to turn down a link which is working and which was being turned up. The situation in which this can happen was described earlier. We want to send this message since bringing a link down could conceivably cause a partition of the network into to components. We must make sure that each disconnected component is aware that the link is being brought down.

Types of Messages

UPDATE(U, c) : contains a table of status changes, and the number of the controlling node for this version of the algorithm

UPDATE_OVER(U, c) : like UPDATE above, but tells a node to terminate the algorithm

BRING_LINK_UP: instructs a node to start an update to turn up the link on which it was received.

BRING_LINK_DOWN: instructs a node to start an update to turn down the link on which it was received.

We now consider the procedures which are followed at each node n in response to a link status change, or receiving an algorithm message. We begin with the initialization which is done when a node processor is first started. The node is not running the algorithm, and assumes all of its adjacent links are down.

Initialization: performed when node n's processor is rebooted

$s(j)$:= DOWN $\forall$ adjacent links j

$T(j)$:= DOWN $\forall$ links j

c := NONE

r := NONE

When a node detects that an adjacent link has become operational, it simply informs the other end node of this. This is the first step in the coordination procedure which will result in an update being started only by the end node with the higher node number. Of course, we only try to turn a link up when no down changes are occurring.

Event 1: $s(l)$ changes from DOWN to UP

(link l becomes operational)

if $U(j) \neq$ DOWN $\forall$ links j **and** $U(l) =$ NO_CHANGE

send BRING_LINK_UP on link l

end

When a node detects the failure of an adjacent link l, or when it is instructed to bring a link down, it usually wants to start an update containing $U(l) =$ DOWN. If a node is not changing the status of link l and lists it as up in its topology table, then it certainly wants to start an update. If the link is being turned up, we want to start a down update

to counter this. In all other cases, starting an update is not necessary either because one is already in progress or because one has already been completed.

Event 2: $s(l)$ changes from UP to DOWN

(link l becomes non-operational)

Event 3: BRING_LINK_DOWN received on link l

if [$U(l)$ = NO_CHANGE **and** $T(l)$ = UP] or $U(l)$ = UP

$U(l) :=$ DOWN

$U(j) :=$ DOWN $\forall\ j : U(j) =$ UP **and** $T(j) =$ DOWN

execute *start_new_version*

execute *check_ack*

end

When a node n receives BRING_LINK_UP from n' over link l, its actions depend on the node number of itself and its neighbor. As mentioned earlier, a link is only brought up if no down updates are present. If node n has a lower node number than n', all that is needed is to send BRING_LINK_UP back to n'. This will cause n' to start an update. If n has the higher node number, then it starts a new algorithm version. The specific functions involved will be discussed shortly.

Event 4: BRING_LINK_UP is received on link l

(connecting nodes n and n')

if $U(j) \neq$ DOWN $\forall$ links j **and** $U(l) =$ NO_CHANGE

if $n < n'$

send BRING_LINK_UP on link l

else

$U(l) :=$ UP

(check for a reconnect)

if *node_connect*$(n, n', T) =$ NO

$U(j) :=$ UP $\forall\ j : T(j) =$ UP

execute *start_new_version*

execute *check_ack*

end

The most complex part of the algorithm is its response to receiving an update message from a neighbor. Most of the complexity is contained in the tests used to determine the appropriate response to a given message. Assume that node n receives an update message containing U' and c' from node n'. First we construct the combined update list according to the rules discussed earlier. We must check to make sure that failed links are not being turned up; and inform our neighbors if we are turning down any working links. If the update list that node n received is exactly the same as the one that it had, then its actions are dictated by the controlling node numbers. The decisions involved were discussed earlier. If the received update has a higher version then n knows about, then node n should propagate the received algorithm version. If the received update list provides n with no new information, and was not an acknowledgement, then it is ignored. Finally, if U and U' combine to form an update list different from both, then n must start a new version of the algorithm, as was discussed.

Event 5: UPDATE(U', c') received on link l

(connecting n and n')

(combine received U table with node n's current U table)

$U^*(j) := \max[U(j), U'(j)] \ \forall$ links j

(make sure that node n is not trying to turn up any failed links)

for each adjacent link j

if $U^*(j) =$ UP and $s(j) =$ DOWN

$U^*(j) :=$ DOWN

(if node n is turning down a working link, it should tell the other end node)

send BRING_LINK_DOWN on each adjacent link j :

$U(j) \neq$ DOWN and $U^*(j) =$ DOWN

if $U(j) =$ DOWN for some link j

$U^*(i) :=$ DOWN $\forall\ i : U(i) =$ UP **and** $T(i) =$ DOWN

else if *node_connect*$(n, n', T) =$ NO

$U^*(j) :=$ UP $\forall\ j : T(j) =$ UP

(tests for how to respond to the received message)

if $U(j) = U^*(j)\ \forall$ links j **and** $U'(j) = U^*(j)\ \forall$ links j

(the received message has not changed node n's update list

and is identical to it)

if $c' > c$

(this version has a higher controlling node

so we wish to propagate it)

$c := c'$

execute *propagate_new_version*(l)

if $c' = c$

(received message is an acknowledgement)

$L(l) :=$ NOT_WAITING

if $c' < c$

(received message is obsolete, so ignore it)

end

else if $U'(j) = U^*(j)\ \forall$ links j

(received message contains all changes that n knows about,

so propagate the received version)

$U(j) := U^*(j)\ \forall$ links j

$c := c'$

execute *propagate_new_version*(l)

else if $U(j) = U^*(j)\ \forall$ links j

(the received message contained no new changes,

and was not an acknowledgement, so ignore it)

end

else

(node n has formed a new update list which is different from
both U and U', so start a new version)

$U(j) := U^*(j)$ $\forall$ links j

execute *start_new_version*

execute *check_ack*

end

When an UPDATE_OVER message is received, node n checks to see if the message refers to the version of the algorithm which n is running. If it does, then node n can terminate FTA. If it does not, then this means that more changes have occurred since the UPDATE_OVER message was generated, and the received message should be ignored.

Event 6: UPDATE_OVER(U', c') received on link l

if $c = c'$ and $U(j) = U'(j)$ $\forall$ links j

execute *terminate*

end

When a node starts a new version, it makes itself the controlling node and has no respond link. It must then decide which of its adjacent links are involved (used) in this version of the algorithm. FTA uses all links which are up and are not being turned down, and those which are being turned up.

Procedure: *start_new_version*

$c := n$

$r :=$ NONE

$L(j) :=$ NOT_WAITING $\forall$ adjacent links j

$L(j) :=$ WAITING $\forall$ adjacent links j : [$T(j) =$ UP and
$U(j) =$ NO_CHANGE] or $U(j) =$ UP

send UPDATE(U, c) on all adjacent links $j : L(j) =$ WAITING

return

Propagating the current version of an algorithm is nearly identical to starting a

deal with its most recent failure.

2) A down update could have been started about some link. As discussed earlier, this would stop any other links from being turned up.

In either situation node n's response is to transmit BRING_LINK_UP on each working adjacent link which has been turned down. The effect of this is to start an update which will turn these links up. In this way, an update which contains the correct status of these links will eventually be installed by each node.

Procedure: *terminate*

send UPDATE_OVER(U, c) on all adjacent links j :

$[T(j) = \text{UP}$ and $U(j) = \text{NO\_CHANGE}]$ or $U(j) = \text{UP}$

$T(j) := \text{DOWN} \; \forall$ links $j : U(j) = \text{DOWN}$

$T(j) := \text{UP} \; \forall$ links $j : U(j) = \text{UP}$

$T(j) := \text{DOWN} \; \forall$ links $j : \mathit{link\_connect}(n, j, T) = \text{NO}$

$U(j) := \text{NO\_CHANGE} \; \forall$ links j

$r := \text{NONE}$

$c := \text{NONE}$

(check for corrections to the new topology)

send BRING_LINK_UP on all adjacent links $j : s(j) = \text{UP}$ and $T(j) = \text{DOWN}$

End of Algorithm. ∎

The justification that FTA works correctly is quite long and complex. For that reason, it has been placed in an appendix.

3.3 Analysis of FTA

Although FTA is a complex algorithm to describe, it is comparatively simple to analyze. We begin with the most important measures of goodness: the time and communication complexities for single link topology changes. We consider a network with l links and n nodes.

The communication complexity in the case of a single link topology change depends

new version except that node n has a respond link, and is not the controlling node.

Procedure: *propagate_new_version*

$r := l$

$L(j) :=$ NOT_WAITING $\forall$ adjacent links j

$L(j) :=$ WAITING $\forall$ adjacent links $j : [T(j) =$ UP **and**

$U(j) =$ NO_CHANGE] **or** $U(j) =$ UP

$L(r) :=$ NOT_WAITING

send UPDATE(U, c) on all adjacent links $j : L(j) =$ WAITING

return

The procedure used to see if all acknowledgements have been received is very straight forward. There are two cases depending on whether a node has a respond link (i.e. is the controlling node).

Procedure: *check_ack*

if $L(j) =$ NOT_WAITING $\forall$ adjacent links j

if $r =$ NONE

execute *terminate*

else

send UPDATE(U, c) on link r

return

When a node terminates the algorithm it first propagates the UPDATE_OVER message which it received. It then installs its new topology table with all of the changes specified in the update list U. The node must also turn down any links that it is now disconnected from.

The one remaining task for a node to perform, after changing its tables, is check if this this FTA version has turned down (or never turned up) any of its adjacent links which are now working. This can occur in two ways:

1) The link could have become operational while an update was still in progress to

on whether the link is being brought up or down. When a link is brought up, the use of BRING_LINK_UP messages insures that exactly one version of the algorithm is started. It has been explained earlier that the propagate – acknowledgement portion of the algorithm sends exactly one message in each direction on each link. This gives $2l$ messages for this portion. In the final flood, during termination, each node, except for the controlling node, sends a message on all but one of its adjacent links. The controlling node sends a message on each of its adjacent links. This gives a total of $2l-(n-1)$ messages for this portion of the algorithm. Combining results, we get a tight best case bound on the number of messages sent for a single link topology change:

$$C_{FTA} > 4l - (n-1)$$

When a link is brought down, two versions of the algorithm may be started, one by each end node of the failed link. If we assume that these two end nodes are still connected after the link fails, then one algorithm version will dominate and be completed. Before this happens, it is possible for the other version to have nearly completed its propagate – acknowledgement phase. This is illustrated in Figure 3.3.1. Assume node 1 very quickly detects that link j has failed, whereas node 2 is slow to detect the change. Also assume node 3 is extraordinarily slow in relaying node 1's update message to node 2. It is possible for node 1 to receive acknowledgements from each node on the right before node 2 discovers the change. Now, node 2 can discover that link j has failed and start a version with a higher controlling node number than the version started by node 1. Node 2 only does this if it discovers that j has failed by itself, before receiving an update message from node 3. Otherwise, node 2 would acknowledge the version started by node 1. The result of this scenario is that when a node fails there can be an extra propagate – acknowledgement phase, but only one termination phase. Therefore, the worst case communication bound for a single link change is:

$$C_{FTA} < 6l - (n-1)$$

In summary, for a single link topology change, we have the following results:

$$4l - (n-1) \leq C_{FTA} < 6l - (n-1)$$

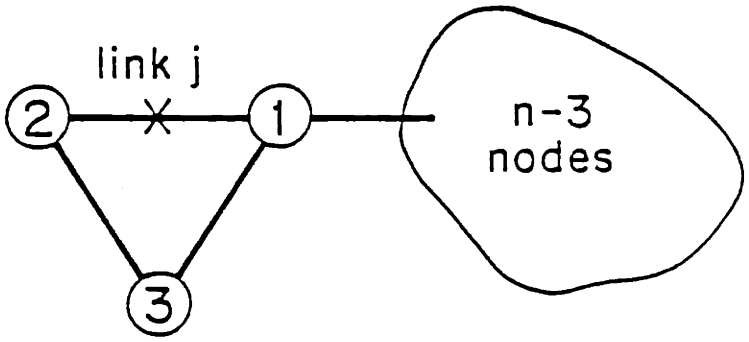

Figure 3.3.1: FTA Communication Complexity Example

$$C_{FTA} \sim O(l)$$

In most situations when a link fails, the timing and topology make the worst case scenario quite unlikely, and we can expect C_{FTA} to be closer to its lower bound. When a link is brought up the lower bound is always met with equality.

The time complexity for a single link topology change is trivial to compute. We consider the time complexity to be the time to completion of the algorithm after each end node has detected the change. It is impossible for one version of the algorithm to be slowed down by the existence of another version with a lower version number. Any time there is more than one message queued for transmission on a link, there is no need to send the earlier message. The later message must have a higher version number which will render the earlier message obsolete. Therefore, we can restrict our attention to the version of the algorithm which dominates. It has three phases (propagate, acknowledge, and terminate) each of which is time bounded by the diameter of the network. Since the network diameter is upper bounded by $n-1$, the time complexity bound is:

$$T_{FTA} \leq 3(n-1) \qquad\qquad T_{FTA} \sim O(n)$$

The situation when multiple topology changes occur is slightly more complicated. However, having many topology changes in a short time is so rare that we are mainly concerned with showing that bounds on C_{FTA} and T_{FTA} exist and are polynomial. We are not concerned with finding good bounds. To examine the communication complexity, we assume that k status changes occur in a network. From the previous discussion, the number of messages sent by FTA per version is bounded by $4l$. To establish an extremely loose bound, we assume that each node goes through as many versions as possible. Since there are k changes, each node can find out about each one separately. For each change there can be at most n controlling nodes. Therefore the greatest number of versions that a node can go through is kn. Since each node can transmit two messages on each adjacent link for each version, this gives $4lnk$ messages. However, since FTA handles down changes first and then up changes, we need to go through the preceding argument twice. This gives a result of:

$$C_{FTA} < 8nlk \qquad\qquad C_{FTA} \sim O(nlk)$$

Notice that the worst case for C_{FTA} is still polynomial bounded. It is interesting to note that the order of C_{FTA} is the same as it was for the Finn algorithm with multiple changes. However, this bound is extremely loose, and we can expect FTA to use significantly less communication for most ordinary topologies. It was mentioned in Chapter 1 that there is a tendency toward sparsity in the topology of actual networks. In a sparse network, the number of versions of FTA which can start for a given U table is typically far less than the number of nodes in the network. We can expect FTA to perform quite well in when only a few status changes occur in an actual network. This is important since node failures typically involve the failure of a small number of links (usually 1–4). We can expect FTA to perform efficiently for node failures, and at least as well as the Finn algorithm in more complex topology change situations. The exact efficiency of an algorithm in such situations is not a serious concern since the probability of many nearly simultaneous topology changes is so small.

The time complexity of for many topology changes can be defined as the to com-

pletion after the last link status change has been detected by each of that link's end nodes. Many versions of FTA can be started in such situations, but as explained earlier, we can restrict our attention to the one with the highest version number. It can take up to $d \leq n-1$ time units (where d is the diameter of the network) for the version which will go to completion to be formed. The previous bound of $3(n-1)$ still applies to this version after it is formed. However, as with C_{FTA}, we must go through this argument twice: once for the down changes, and once for the up changes. Ignoring the -1's gives the following bound.

$$T_{FTA} < 8n \qquad\qquad T_{FTA} \sim O(n)$$

The time complexity has gone up compared to a single change, but is still $O(n)$.

Overall, FTA is quite efficient for probable topology change events. For such events, the order of its communication is the same as for the ARPANET update procedure. For probable events, FTA maintains the robustness of the Finn algorithm without sacrificing much efficiency. A further application of FTA will be seen in Chapter 4 when it is used to simplify the solution to the routing information problem.

3.4 Shortest Path Topology Algorithm (SPTA)

In many network situations it is not desirable to have an algorithm which exhibits all of the resynchronization properties of the previously discussed Flooding Topology Algorithm. Resynchronization will be used in Chapter 4 to help solve the routing information problem, but it is not a necessary property if all we are concerned with is solving the topology problem as efficiently as possible. Therefore, it is useful to examine algorithms which dynamically maintain accurate network topologies without necessarily having the resynchronization property. We can now consider algorithms which are truly distributed in terms of both their operation and their control. The FTA and kSTRA algorithms are distributed in the sense of operating simultaneously at many processors separated by communications delay. Yet, the controlling steps or decisions of the algorithms require that all information about changes eventually collects at a single node. This "control" node then

takes steps to bring the algorithm to a conclusion. With the resynchronization requirement lifted, there is no need to introduce the concept of a controlling node. Each node can merely respond to topology changes in a consistent manner, on an equal basis with other nodes.

We are still faced with the fundamental problem of how to distinguish between old and new information about the status of a link. FTA solved this problem by limiting the amount of information about a given link that could be present on the network at any time through an acknowledgement scheme. Here, we take a completely different approach. We place no restrictions on the transmission of new information; nodes transmit information about the status of an adjacent link whenever its status changes. Each node keeps, at all times, a shortest hop path length for each link in the network. When conflicting information is received, the information which was received over the shortest path is used. Thus a node determines the relative validity of a message by using a distance (in hops) assigned to the link on which the message is received. This simple scheme results in an algorithm that is remarkably simple and efficient.

Overview of operation

Each node n in the network maintains a topology table T, called its main topology table. T contains an entry $T(j)$ for each link l in the network. These entries reflect node n's current best estimate of the operational status of each link in the network. In addition to its main topology table, each node maintains a port topology table T_j for each port j at the node. These tables are shown in Figure 3.4.1. A node's port topology table for port j is merely a slightly delayed copy of the main topology table of the neighboring node reached via port j. Node n's port topologies are delayed versions of each of its neighbor's main topologies. Therefore, each node keeps track of its own topology as well as those of its neighbors.

In order to maintain the topology tables described above, we must adopt the following rules for the transmission and reception of messages between nodes. The messages we speak of contain a set of modifications to a topology table. In this sense they are analogous

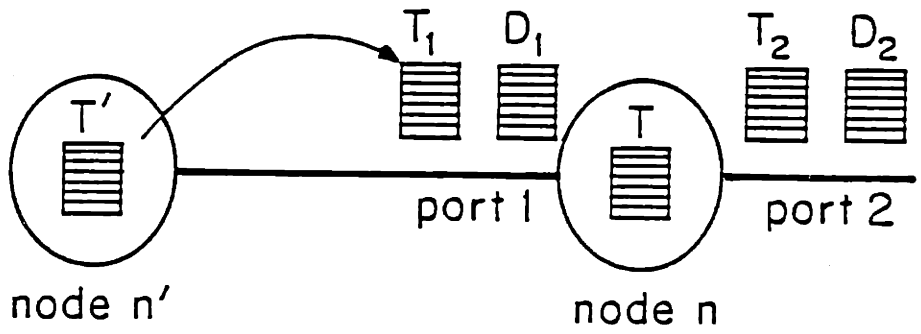

Figure 3.4.1: SPTA Data Structures at node n

to the update messages of FTA, but in this algorithm they are immediately installed in a port topology table.

1) If a node's main topology changes, it sends a message describing these changes on each of its operating links.
2) If a node receives a topology change message over a link, it makes the received changes in the port topology table associated with that link.
3) When a node detects that a link has become operational, it transmits its entire main topology table over that link.

The first two rules are a straight forward way of maintaining the topology tables described earlier. The last rule is needed since while a link is down, no topology updates are received over it. When the link comes back up, the port topology at a node can be very out of date. Therefore, we must transmit the entire main topology table to make the associated port topology current. These three rules completely specify the message exchange portion of SPTA.

The only remaining portion of SPTA to describe is how a node decides what its main topology table will contain. It does this by using the distance tables, D_i shown in Figure 3.4.1. SPTA uses shortest hop distance as a metric to judge the relative validity

of topology information. This metric is by no means unique, but is certainly reasonable. Usually, we would expect the most up to date information about the status of a link to first arrive at a node over its shortest path to that link. There are times when this will not be true, and the algorithm will temporarily reach an incorrect conclusion about the status of a link. However, we will show that the correct decision will eventually be made, and that the algorithm is guaranteed to work in the presence of arbitrarily complex topology changes.

We now discuss in detail the construction of the distance tables, D_i. Consider port i of node n with a port topology table T_i. Node n's neighbor via port i is node n'. We would like each entry $D_i(l)$ in the distance table to be a measure to how likely node n is to believe the topology information in $T_i(l)$. Let link i be the link connecting n and n'. We make $D_i(l)$ equal to the number of hops in the shortest path from node n to link l which uses link i.† The path is computed using the port topology T_i. This is equivalent to the length of the shortest path from node n' to link l, plus 1. However, there are two exceptions to this. If link l is adjacent to node n we always want $D_i(l) = 0$. This reflects the fact that node n always knows the current status of link l. If according to topology T_i, node n's distance to link l is smaller than node n''s distance to link l, then we want $D_i(l) = \infty$. The reason for this is that if a node is closer to a given link than its neighbor, it does not want to listen to that neighbor about the status of the link. This is reflected in the choice of a very large distance value.

The method for computing D_i just described can be summarized. We define the function $shortest\_path(n, l, T_i)$ to be the length in hops of the shortest hop path from node n to link l according to topology T_i. Node n's neighbor via port i is node n'. Node n computes its port topology table for each port i as follows.

$$D_i(l) = \begin{cases} shortest\_path(n', l, T_i) + 1 & \text{for } shortest\_path(n, l, T_i) \geq shortest\_path(n', l, T_i) \\ 0 & \text{for } shortest\_path(n, l, T_i) = 0 \\ \infty & \text{otherwise} \end{cases}$$

† The distance from a node to a link is defined as the smaller of the distances from the node to each of the link's end nodes.

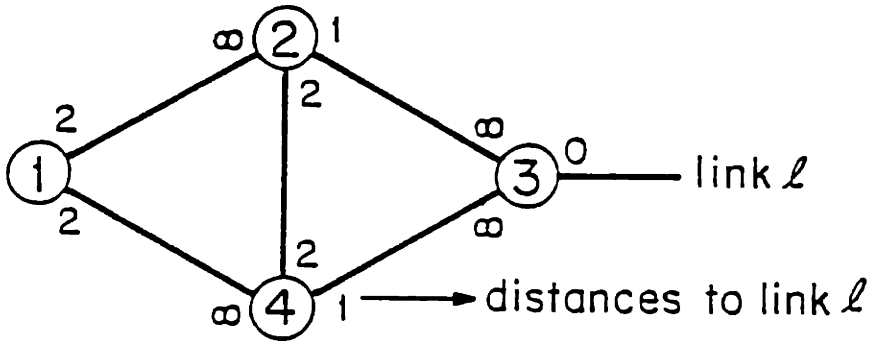

Figure 3.4.2: SPTA Port Distance Table Example

An example of these computations is shown in Figure 3.4.2. We assume that the topology is constant and all topology tables are the same. We have entered beside each port its distance table entry for link l. Notice the use of 0 and ∞ as described above.

Having computed its port distance tables, a node can then construct its main topology table, T. To choose its main topology entry $T(l)$ for a link l, node n compares the distance table entries for link l contained in each of its port distance tables. It selects the port with the minimum distance to link l and enters that port's topology table entry for link l in its main topology table. This can be expressed as follows.

$$T(l) = T_{\underset{i}{\text{argmin}}\{D_i(l)\}}(l)$$

An ambiguous case results when two ports have the same distance to a link l. An example of this is shown in Figure 3.4.2. Node 1 has two 2-hop paths to link l. surprisingly, what SPTA does in this case is nearly arbitrary. The algorithm will work correctly if the following rule is satisfied.

> Let K be the set of ports with minimum distance to some link l. If $T_i(l) =$ UP for each $i \in K$, then a node must set $T(l) =$ UP. Likewise for all $T_i(l) =$ DOWN.

Whenever there are discrepancies in the $T_i(l)$'s a node can make an arbitrary choice. Subject to the above constraint, the rule used to break ties does not effect the correctness of SPTA. However, the rule does effect the Communication complexity for multiple failures. This will be discussed in section 3.5. In the presentation of SPTA which follows, we adopt the rule that each message received on a shortest path is installed in the main topology.

Having discussed how a node forms its main topology, we have completely described the theory behind SPTA. To put this algorithm into practice a node does the following when a link status change is detected.

1) The appropriate port topology is modified.
2) That port's distance table is recomputed.
3) The main topology is recomputed.
4) If the main topology has changed, a message describing the changes is sent on each working link.

We now present a pseudo-code description of SPTA. This particular presentation uses the Dijkstra algorithm [14] to compute shortest paths. The particular shortest path algorithm used is arbitrary, but Dijkstra's algorithm is particularly well suited since our link distances are all positive (in fact, they are all equal to one). For convenience, each node also keeps a main distance table, D. The entries in this table obey the following rule:

$$D(l) = \min_i\{D_i(l)\} \ \forall \text{ links } l.$$

Data Structure Definitions: contained at each node n

$s(j)$ = the DLC status for link j

$T(j)$ = node n's main topology table entry for link j

$D(j)$ = shortest hop path length to link j

$T_l(j)$ = port l's topology table entry for link j

$D_l(j)$ = shortest path length to link j that uses link l (except as noted)

$C(j)$ = whether $T(j)$ has just changed

$L(j)$ = the port over which node n believes topology information about link j
i.e. $T(j) = T_{L(j)}(j)$

J = the set of links which have changed status

Used in Dijkstra's Algorithm:

$d(j)$ = path length of link j

$N(i)$ = node n's shortest path distance to node i

$P(i)$ = whether a node is permanently labeled in Dijkstra's Algorithm

Notes:

- We refer to a link by a link number, such as: $T(j)$, or by a pair of incident nodes, such as: $T(i,k)$, whichever is more convenient.
- Primed variables refer to information received from neighboring nodes.

Message Format: message from n' to n on link l.

$$U([\overbrace{j,}^{\text{link}} \underbrace{T_l'(j)}_{\text{status}}], [k, T_l'(k)], \ldots)$$

This updates port l's topology, for link j, at node n to $T_l'(j)$ etc.

To initialize a node, we assume that all the links in the network are down. Therefore, a node's distance to all non-adjacent links is ∞.

Event 1: Initialization: n's node processor is rebooted

$s(j) :=$ DOWN $\forall$ for all ports j

$T_i(j) :=$ DOWN $\forall\, j, i$ (for all links and ports)

$D_i(j) := \begin{cases} 0 & \text{for } j = i \\ \infty & \text{otherwise} \end{cases}$ $\forall$ ports i

$T(j) :=$ DOWN $\forall$ links j

$D(j) := \begin{cases} 0 & \forall\, j : \text{link } j \text{ is incident on node } n \\ \infty & \text{otherwise} \end{cases}$

$L(j) :=$ arbitrary, one of node n's ports

When a link becomes operational, a node follows the four steps mentioned earlier. It must also send its entire topology table on the now operational link.

Event 2: Link l between nodes n and n' is now operational

($s(l)$ changes from DOWN to UP)

$J := \{l\}$

$T_l(l) :=$ UP

execute *shortest_path*(l)

execute *compute_topology*(l, J)

send $U([j, T(j)], \ldots \ \forall\ j : C(j) =$ CHANGE) on all links except link l

send $U([j, T(j)], \ldots \ \forall\ j)$ on link l

When a link fails, we follow the same four steps as before. Now, however, the task of computing the new port distance table is easy. Since the link is down, its port distance table has ∞ entered for each link except for the adjacent one which has failed. The distance to adjacent links is always kept equal to 0.

Event 3: Link l between nodes n and n' is now non-operational

($s(l)$ changes from UP to DOWN)

$J := \{l\}$

$T_l(l) :=$ DOWN $\forall$ links j

$D_l(j) := \begin{cases} 0 & \text{for } j = l \\ \infty & \text{for } j \neq l \end{cases}$

execute *compute_topology*(l, J)

send $U([j, T(j)], \ldots \ \forall\ j : C(j) =$ CHANGE) on all links

When an update message is received, we again follow the four steps.

Event 4: $U'([j', T_l'(j')], \ldots)$ received on link $l = (n, n')$

$J := \{\text{all links in message } U'\}$

$T_l(l) := T_l'(j) \ \forall \ j \in J$

execute *shortest_path*(l)

execute *compute_topology*(l, J)

send $U([j, T(j)], \ldots \ \forall \ j : C(j) = \text{CHANGE}\ )$ on all links

To compute a node's port distance table, we apply the rule specified earlier. Node n first computes the distances from itself. It then computes the distances from its neighbor, n'. The rule discussed earlier is then used to create D_l. A standard implementation of the Dijkstra algorithm is used to compute shortest paths.

Procedure: *shortest_path*$(l = [n, n'])$

$$d(j) := \begin{cases} 1 & \text{for } T_l(j) = \text{UP} \\ \infty & \text{for } T_l(j) = \text{DOWN} \end{cases}$$

$$N(i) := \begin{cases} 0 & \text{for } i = n \\ \infty & \text{for } i \neq n \end{cases}$$

$dijkstra(N, d)$

$self\_dist(i, k) := \min\{N(i), N(k)\} \quad \forall \quad \text{links } (i, k)$

$$N(i) := \begin{cases} 0 & \text{for } i = n' \\ \infty & \text{for } i \neq n' \end{cases}$$

$dijkstra(N, d)$

$neigh\_dist(i, k) := \min\{N(i), N(k)\} \quad \forall \quad \text{links } (i, k)$

$$D_l(j) := \begin{cases} \infty & \forall \ j : self\_dist(j) < neigh\_dist(j) \\ neigh\_dist(j) + 1 & \text{otherwise} \end{cases}$$

$D_l(l) := 0$

return

Procedure: $dijkstra(N, d)$

$P(i) :=$ NO $\forall$ nodes i

loop:

if $P(i) =$ YES $\forall$ nodes i

return

$i = \underset{k:P(k)=\text{NO}}{\text{argmin}} \{N(k)\}$

if $N(i) = \infty$

return

else

$P(i) =$ YES

$N(k) = \min_{k:P(k)=\text{NO}} \{N(k), N(i) + d(i,k)\}$

go to loop

The procedure for computing a node's main topology is more complicated than expected considering the simplicity of the rule for forming T. The main difficulty is due to the algorithm messages containing information about several links. This creates a difficulty in the rule used to break ties. Assume that node n receives a message about link l over link i. Let link i be one of several adjacent links which lies on a shortest path to link l. Node n wants to install this message in its main topology. However, there may be other links for which port i is also on a shortest path. Node n does not want to change which port topology is being listened to for these links. Set J is used to keep track of which links messages have been received about, and is helpful in keeping track of when to switch port assignments, $L(j)$. Node n must also keep track of which entries in its main topology have changed. This is done by C. One final point is that a node must never listen to messages about a link j to which it is not connected (i.e. all $D_k(j) = \infty$).

Procedure: *compute_topology*(l, J)

for all links j :

$C(j) :=$ NO_CHANGE

$D(j) := \min_k \{D_k(j)\}$

if $D(j) < \infty$

if $D_l(j) = D(j)$ **and** $j \in J$

$L(j) := l$

else if $D_{L(j)}(j) \neq D(j)$

$L(j) := \text{argmin}_k \{D_k(j)\}$

if $T(j) \neq T_{L(j)}(j)$

$T(j) := T_{L(j)}(j)$

$C(j) :=$ CHANGE

return

End of Algorithm. ∎

3.5 Correctness Proof of SPTA

To prove that SPTA works properly, we must show that it meets the following three requirements when faced with an arbitrary set of topology changes.

1) The algorithm can be started asynchronously by any node at the site of a link topology change.
2) If topology changes stop occurring for a sufficiently long but finite time, the algorithm must terminate.
3) Let N be a maximum connected set of nodes. When the algorithm terminates, each node $n \in N$ will know the correct status of each link which is incident on at least one member of N.

We define the algorithm to be "started" or "running" when one or more updates are being transmitted on a network link, or are being processed by a network node. The algorithm has terminated when it is no longer running. For the purposes of analyzing the algorithm,

it is useful to view the transmission time of an update message to extend until it has arrived at its destination node, and has been processed. Thus when an update message arrives, we consider it to have an instantaneous effect.

The first requirement is that the algorithm can be started asynchronously by a node at the site of a link status change. Since a node transmits update messages immediately upon detecting a status change on one of its adjacent links, this requirement is true by inspection of the algorithm.

We begin the proof by assuming that the network is in steady state. By this we mean that each node in a maximum connected set of nodes has recorded in its topology the correct status for each link incident on a member of the set. By "correct" we mean that the recorded status reflects the true operating state of the link. This assumption does not restrict the proof since it is valid for a node which has just completed the initialization phase of the algorithm. After initialization, a node is itself a maximum connected set. From this steady state condition, some arbitrary set of link status changes occur over a period of time. We require that there must eventually be a sufficiently long but finite time interval during which no further status changes occur. At the end of this interval, we wish to show that a steady state condition has been reestablished. This new steady state may involve different maximum connected sets if parts of the network are separated or joined.

A set of L status changes, or updates will be denoted $S = \{s_1, \ldots, s_L\}$. If a single link has more than one status change reported by one or both of its incident nodes, only the latest one is considered to be in the set since it is this update that we wish to show is correctly recorded at each connected node.

At all times, each node maintains a main distance table D, which lists the length of a path from itself to each link. During steady state, each such path is a shortest path; however when an update is in progress, it may be outdated. At a node, call each incident link l for which $D_l(j) = D(j)$ a "preferred" link for updates received about link j. In steady state, node n's preferred links for link j are those adjacent links which lie on a shortest path

from n to j. If there are multiple shortest paths, there may be multiple preferred links. When a node receives an update about link j over a preferred link for link j, the update will be installed in node n's main topology table. This can be seen by examining the second "if" statement in procedure *compute_topology*. We refer to an update received over link l as being installed when $L(j) = l$, and thus $T(j) = T_l(j)$. At all times, node n has installed in its main topology an update for link j which was received over one of n's preferred links for link j. The concept of a preferred link is useful in proving that the algorithm works correctly.

In the proof which follows, we assume that a network consisting of one or more maximum connected sets of nodes is in steady state at time t_s. Let T^* be the steady state topology that an omniscient observer would construct from examining the network. At each node n, $T^*(j) = T(j)$ for all links j which are connected to n. After time t_s, a set of L status changes, $S = \{s_1, \ldots, s_L\}$ occur concerning links $\{l_1, \ldots, l_L\}$. If a link changes status more than once, we consider only the last change to be in S. Each change, s_i, of link l_i, is eventually detected by both end nodes of link l_i. When this occurs, we call it time t_0. From time t_0 until some later time t_f, no further status changes occur. Provided that the finite time interval $t_f - t_0$ is long enough, we wish to show that at time t_f steady state has been reestablished, and the algorithm has terminated. Let T' be the topology which results from making the above L status changes to T^*. At time t_f we wish to show that at each node n, $T'(j) = T(j)$ for all links j which are connected to n. In the proof, when we say that "node n knows the status of link j" we mean that $T(j) = T'(j)$ at node n.

We begin by showing a few Lemmata useful later in the proof. These Lemmata depend on the DLC assumptions in section 3.1.

Lemma 1. *Let l be a link connecting nodes n and n'. If l is operating and there are no updates concerning a link j traveling on l from n' toward n, then $T_l(j)$ at node n equals $T(j)$ at node n'.*

Proof. Recall from procedure compute topology that any time a node's main topology

changes, it sends an update on each of its operating links. Assume that link l has just become operational. Let s_0 be node n''s status for link j when it first detected that link l was operating. Node n' sends $T(j) = s_0$ to node n. When s_0 arrives at n, node n immediately sets $T_l(j) = s_0$. At that moment either $T_l(j)$ at node n equals $T(j)$ at node n', or node n' has a new status, s_1, for link j. If n' does have a new status, it must have sent an update $T(j) = s_1$ on link l. Therefore, at any time, node n either knows node n''s latest status for link j, or n' has sent an update announcing this latest status on link l. ▌

Using T' an omniscient observer could calculate the shortest hop path from node n to each network link. When we refer to a link l as being j hops from n, we mean that the length of the shortest hop path from link l to node n, using T' is j hops long. The shortest path length from n to a link to which n is not connected is considered to be ∞.

Define a time t_j at which each node knows the correct status of each link $\leq j$ hops from itself. Except for t_0, we have not yet shown that such a time will ever exist. Assume that from t_j until t_f each node's status about a link $\leq j$ hops away cannot change. Also, define t_j' as a finite time after t_j at which each update that each node transmitted at a time $\leq t_j$ has been received at its destination. These definitions are useful in the following lemmata.

Lemma 2. (This lemma is not explicitly needed in the proof, but it contributes to a conceptual understanding of how the algorithm works.) *At time t_j, defined earlier, using the main topology T at a node n we can calculate all shortest paths to each link l that is $\leq j+1$ hops from n. Furthermore, these shortest paths cannot change before time t_f.*

Proof. All shortest paths between n and l must involve only links which are $\leq j$ hops from n. Since n knows the correct status of each of these links (i.e. $T(\cdot) = T'(\cdot)$ for each of them), we can calculate all shortest paths from n to l using T. If the status of each link $\leq j$ hops from n cannot change before t_f, then the shortest paths calculated using these status' cannot change before t_f. ▌

Lemma 3. *A time t_j', defined earlier, each of node n's preferred links for each link l that is $j+1$ hops from n must lie on a shortest path from n to l. Furthermore, these preferred links cannot change before time t_f.*

Proof. Let k be a link joining nodes n and n'. At t_j' there are no updates on k traveling toward n which concern a link m that is $\leq j$ hops from n'. By lemma 1, $T_k(m)$ at node n must equal $T(m)$ at node n'. Since $T(m)$ at node n' is correct by assumption, $T_k(m)$ is also correct for each link m that is $\leq j$ hops from n'.

Let l be a link that is $j+1$ hops from n. If there is a shortest path from n to l that uses link k, then this path must use only links which are $\leq j-1$ hops from n'. Since T_k at node n contains the correct status for each such link, procedure shortest path must find $D_k(l) = j+1$. Conversely, if there is no shortest path from n to l using link k, $D_k(l)$ must be $> j+1$. If it were not, there would be a $\leq j+1$ hop path from n to l, using link k, that consisted entirely of links that were $\leq j-1$ hops from n'. By assumption no such path exists. Therefore, each link k adjacent to n for which $D_k(l) = j+1$ must lie on a shortest path from n to l. Also, since no path shorter than $j+1$ hops exists from n to l, each link k for which $D_k(l) = j+1$ must also be a preferred link. Therefore, each link k for which $D_k(l) = j+1$ must be a preferred link and must lie on a shortest path from n to l.

Since the topology table entries used to calculate each preferred link cannot change before t_f, neither can the preferred link. ∎

Theorem 1. *A finite time after t_0 defined earlier, each node n knows the correct status for each link to which it is connected, according to T'. Each such status will not change before t_f.*

Proof. (by induction)

Basis step: At time t_0 each node knows the status of each of its adjacent links, and this cannot change before t_f. Equivalently, each node n knows the status of each link ≤ 0 hops away: $T(j) = T'(j)\ \forall\ j \leq 0$ hops from n.

Assume that there is a time t_j, defined earlier, at which each node knows the status of each link $\leq j$ hops away, and that this status cannot change before t_f. Then, we wish to

show that a finite time after t_j there must be a similarly defined time t_{j+1}.

Let t_j' be a finite time after t_j at which each update transmitted before t_j has been received. Consider a link l that is $j+1$ hops from n. By lemma 3, each of node n's preferred links for link l must lie on a shortest path from n to l. Let link k connecting n and n' be one of n''s preferred links for link l. Since k is on a shortest path from n to l, if n is $j+1$ hops from l, then n' must be j hops from l. By assumption, at time t_j' there are no updates traveling from n' to n concerning link l. Therefore, by lemma 1, $T_k(l)$ at node n equals $T(l)$ at node n'. By assumption $T(l)$ is correct since l is j hops from n'. Thus if k is a preferred link for link l, then $T_k(l)$ is correct. Examining procedure *compute_topology* we can see that $T(l)$ at node n is always equal to $T_k(l)$, where k is one of n's preferred links for link l. Recall that there are no updates about l traveling on k. Therefore, at t_j', $T(l)$ must be correct for each link l that is $j+1$ hops from n. By lemma 3, n's preferred links,for each link l, cannot change before t_f. By assumption, $T_k(l)$ cannot change before t_f, and thus $T(l)$ at node n cannot change before t_f. Then at time t_j each node knows the correct status of each link $\leq j+1$ hops away, and this status cannot change before t_f. We refer to time t_j' as time t_{j+1}.

By induction on j, at a finite time after t_0, each node n must know the correct status of each link which is a finite distance away from n. Links that are a finite distance away from n are those to which n is connected according to T'. ▌

Theorem 2. *A finite time after theorem 1 is satisfied, the algorithm must terminate.*

Proof. When theorem 1 is satisfied, no further topology changes are installed about links to which a node is connected. Examining procedure *compute_topology*, we can see that topology changes are never installed about links to which a node is not connected, after theorem 1 is satisfied. Once theorem 1 is satisfied, no further changes are made to each node's main topology before t_f. Therefore, no updates are generated after theorem 1 is satisfied. If we wait a finite time after this, any updates present in the network will have been received, and the algorithm terminates. ▌

Together, theorems 1 and 2 prove that SPTA works correctly.

3.6 Analysis of SPTA

The analysis of the time and communication complexity of SPTA will parallel the analysis of FTA in section 3.3. We begin by analyzing the most important case, that of a single link topology change. The communication complexity of the algorithm in this case is quite easy to compute. Each node receives a message over a shortest path to link l and modifies its main topology. When a node modifies its main topology, it sends one message on each of its adjacent links. Therefore, one message is sent on each port. This gives a communication complexity of:

$$C_{SPTA} = 2l \qquad\qquad C_{SPTA} \sim O(l)$$

for single link topology changes.

We assumed in the above analysis that if a link is being turned up, it does not reconnect two components. Reconnects may require $4l$ messages, since each node may change its main topology twice: once for the repaired link, and once to adopt the correct topology for the adjacent component. In addition, half of these messages have $O(l)$ complexity since they contain information about each link in a given component. SPTA can be easily modified to have each node at a component border delay broadcasting a message about a repaired link until it has received the neighboring component's topology. This would make the communication complexity $2l$ for all single topology changes(neglecting message size). This modification was not made to the version of SPTA presented in section 3.4 since it contributes little to understanding how the algorithm works, and creates a more cumbersome presentation.

The time complexity of the algorithm can be found by examining the proof in section 3.5. We are interested in the time to completion after the last link status change has been detected by each end node. As with FTA, we can neglect any queueing delay in transmitting messages on a link since it is never necessary to have more than one message awaiting transmission. When a new message is generated, it can be combined with an older

message which is waiting to be transmitted. The newer message's status change for a given link renders any older status changes about that link obsolete.

Examining the proof of theorem 1, we can see that there is a time t_j at which each node knows the correct status of all links $\leq j$ hops away. The time between t_j and t_{j+1} is the maximum transmission time for messages on the network. When computing time complexity, this is assumed to be 1. Therefore, after j time units a node knows the correct status of each link $\leq j$ hops away. If the network has diameter $n-1$, after $n-1$ time units each node knows the correct status of each link to which it is connected. We must then wait one more time unit for any messages still traveling on the network links to arrive. This gives a time complexity of:

$$T_{SPTA} \leq n \qquad\qquad T_{SPTA} \sim O(n).$$

This result is valid for arbitrarily complex topology change events. Unlike FTA, the time complexity of SPTA does not increase when multiple status changes occur.

The last quantity of interest in analyzing SPTA is its communication complexity when multiple status changes occur. We again neglect the effects of component reconnects. Assume that there are k status changes in the network. We would expect each node, in the worst case to find out about each change individually. This would cause each node to transmit k messages on each adjacent link, for a total of $2lk$ messages sent on the network. In nearly all cases, this is a valid bound on the number of messages sent.

There is one type of scenario in which the above bound is not valid; it is best illustrated by an example. Refer to Figure 3.6.1. We are concerned with node 1's view of the status of link A. Initially, let links A and B both be operating. Node 1's shortest path to link A is path 1. Now assume that link A fails. Node 2 immediately sends a message containing "link A has failed" over path 1 and path 2. Let node 1 receive the message over path 1 first. Since path 1 is its shortest path to node 1, it immediately modifies its main topology and considers link A to be down. Now let link B fail. Node 1 detects this

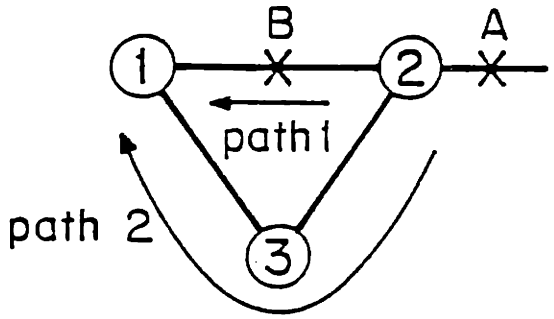

Figure 3.6.1: Multiple Link Failure Example

failure and makes path 2 its shortest path to link A. Therefore, the link A entry in its port topology for link (1,3) will now be stored in its main topology. If the message containing "link A has failed" has not yet arrived on path 2, node 1 will decide that link A is up since this is the last message that it has received over path 2. When the "link A has failed" message arrives on path 2, node 1 will once again decide that link A is down. The algorithm will then terminate with each node knowing the correct topology.

In the above example, node 1's main topology changed three times for link A whereas the status of link A actually changes only once. If the timing is chosen just right when multiple failures occur, this "cycling effect" can result. This makes it difficult to bound the communication complexity of SPTA. Essentially, what happens is that a second failure can cause a node to change its shortest path to the first link that failed. To bound C_{SPTA} we would first like to limit the number of times that a node can change its shortest path to a given link. How many this can happen depends on the rule which SPTA uses to break ties when it has several paths of minimum length. We adopt the rule that when a node has several ports with the same distance to a given link l, it picks one (using any desired rule) and only listens to messages about link l which are received over that port. A node never changes this port unless another port attains a distance to link l which is

strictly less than its current minimum. This rule satisfies the two conditions mentioned in section 3.4 and the algorithm will work correctly.

The above rule is different from the one used in the presentation of SPTA in section 3.4. That rule is perfectly reasonable in ordinary topology situations, but its quickness in installing topology changes makes establishing a communication complexity bound difficult.

Using the above rule, we can establish a loose general bound on the communication complexity. Examine Figure 3.6.2. We have added two nodes to the network of Figure 3.6.1, and are interested in the number of status changes that node 4 can experience about link A. There are now three paths over which node 4 can receive information about link A. Node 4 can not actually distinguish between paths 1 and 2, but a omniscient observer could. Whenever a path change occurs, a status change can be generated since some node switches the port topology that it is listening to. We assume that links A and B fail as before. Therefore, node 1 experiences the same three status changes as in the previous example. Node 1 sends each of these three changes to node 4 which installs them. So far, node 4 has experienced 3 status changes about link A. Now let link C fail. Node 4 switches to path 3, and can experience another status change in the act of switching. It can then receive the same three messages from node 5 as it did earlier from node 1. This results in a total of 7 status changes about link A.

Consider carrying out a Gedanken experiment which involves adding two more nodes to the left of Figure 3.6.2, while keeping the same triangular pattern. Now calculate the number of status changes the left most node experiences about the right most link when all four top links fail in the proper sequence. If we continue this process, a little thought shows that a general formula for the number of status changes, S, in the left most node for this type of configuration is:

$$S \leq 2^k - 1$$

Where k is the number of links that fail.

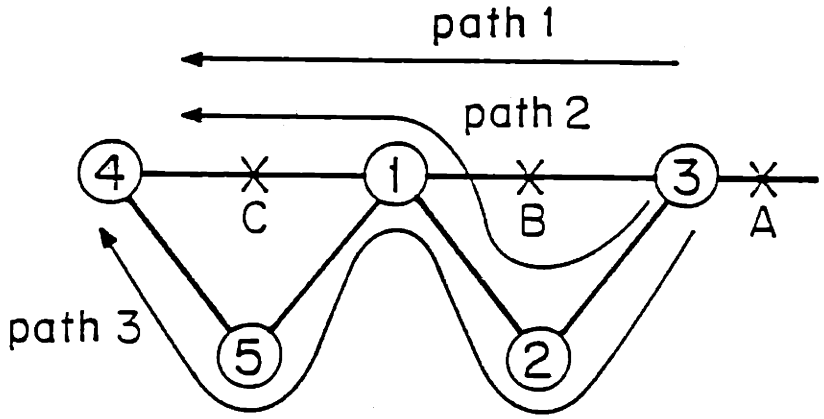

Figure 3.6.2: Another Multiple Link Failure Example

Without presenting a formal proof, we claim that this is the worst case number of status changes about a link that a node can experience when k links fail. This can be justified by noticing that the number of status changes a node experiences is limited by the number of path changes which occur. If there are k failed links, there can be at most k different shortest paths from a node to a failed link. (Recall that when there are multiple minimum length paths, we only consider one to be the shortest path.) In our examples, this is exactly the number of shortest paths from the left most node to the right most link. Therefore, we have actually been looking at the worst case.

To bound the number of messages sent when k links change status, we assume that each node in the network experiences the maximum number of status changes about each link that changes status. Therefore, each node may have at most $k(2^k - 1)$ status changes. Then each node sends at most $k(2^k - 1)$ messages on each link. This gives a communication complexity of:

$$C \leq 2lk(2^k - 1)$$

The above bound is exponential, and this is somewhat discouraging. However, we must remember several important points. The value of k is usually very small. The only

way that cycling can occur is for a shortest path to break after information has been sent over it. This does not happen in the case of a node failure. Therefore, for node failures the communication complexity is $2lk$ where k is the degree of the failed node. A node failure is the most important case where $k > 1$, since the probability of multiple, unrelated, nearly simultaneous link failures is typically very small. The bound is also not needed when links or nodes are repaired. Even in the rare case when several links do fail in a short time, it takes incredibly unfortunate timing for cycling to occur. Therefore, the practical communication complexity of the algorithm is $2lk$, and the above exponential bound is of minor importance.

It can be seen that SPTA is remarkably efficient in terms of both time and communication. It is also quite simple to implement, and is guaranteed to work in all topology change situations.

3.7 Summary of Topology Change Algorithms

We have analyzed 3 existing algorithms for solving the topology problem: ARPANET, Finn, and kSTRA. We have also presented and analyzed two new algorithms for solving this problem: FTA and SPTA. Figure 3.7.1 summarizes the time and communication complexities obtained for single link topology changes. Figure 3.7.2 shows the number of messages that these bounds produce on a network with $n = 62$ and $l = 74$. These were at one time approximate values for the ARPANET. We can see that FTA and SPTA compare very favorably with existing algorithms. In addition, they are event driven, and are guaranteed to work. The other important point to notice is that the Finn algorithm can require an order of magnitude more messages than the other algorithms. This is due to its rebuilding the entire network topology any time that a change occurs.

ALGORITHM	COMMUNICATION COMPLEXITY	TIME COMPLEXITY
ARPANET	$2l$	n
Finn	$(4l+1)(n-1)$	$2n$
kSTRA	$2(n-1)+2l$	$3(n-1)$
FTA	$6l-(n-1)$	$3(n-1)$
SPTA	$2l$	n

Figure 3.7.1: Complexity Comparison of Topology Algorithms (single link topology change)

ALGORITHM	MAXIMUM NUMBER OF MESSAGES
ARPANET	148
Finn	18117
kSTRA	270
FTA	383
SPTA	148

Figure 3.7.2: Maximum Number of Messages Comparison for Topology Algorithms (single link topology change: $n = 62$, $l = 74$)

CHAPTER 4

Algorithms for Broadcasting Routing Information

The routing information problem was described in section 1.4. To review, we are looking for algorithms which are event driven and which only transmit delay measurements which have changed, whenever possible. The desire for efficiency in communications is more acute than with the topology problem since routing delay changes are quite common. We have already discussed one algorithm which solves this problem: the ARPANET update procedure. Other more complicated solutions can be found in [11, 12]. A common method of solving this problem is to take a specific routing algorithm and design a broadcast facility which works for it. We prefer to view the problem as separate from routing, and worthy of independent study. Our algorithms are applicable to routing schemes which require each node to know the delay on each directed link to which it is connected. The fundamental difficulty in solving this problem is the same as for the topology problem: information may be received out of order.

We present two algorithms for solving the routing information problem. The first is a direct extension of SPTA. The second is a set of simple rules which depend on the resynchronization properties of FTA. Both methods guarantee that each node will receive the latest measurement of routing delay about a connected link, and will never confuse earlier messages with later ones. However, they do not guarantee that every delay measurement about a link must be received. For example, if a reconnect occurs, a node in one component will receive only the latest delay measurements contained in the other component, not all of those measurements which have been missed while it was disconnected. This imposes a constraint that the delay measurements must be absolute; they cannot be relative to earlier measurements.

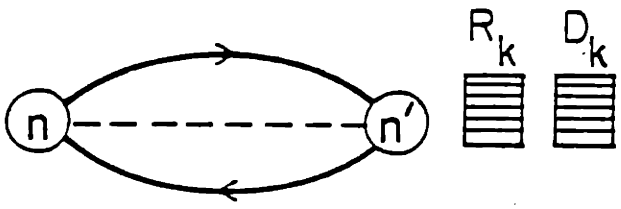

Figure 4.1.1: Directed Links for Routing Information Problem

4.1 A Shortest Path Routing Information Algorithm

The routing information problem can be viewed as a generalization of the topology problem. Consider making the following modifications to the topology problem.

1) Each link is considered to be two directed links. Each node determines the status of its outgoing links.
2) Each directed link has k possible states. One state is non-operational, and $k-1$ are different values for delay.

If we make these modifications to SPTA we obtain an algorithm which is nearly identical to the original. Figure 4.1.1 illustrates some of the modifications. Instead of topology changes which store a binary status for each link, we use routing information tables, R_k which store k-ary statuses for each directed link.

The most significant difference between the extended algorithm and SPTA is in the form of the distance tables, D_k. When we were dealing with undirected links, each end node of a link would detect status changes about it. Now, there is only one "authority" on the status of a directed link: the node from which the link emanates. For example, in Figure 4.4.1 node n reports on the status of link (n, n') whereas node n' reports on the

status of link (n', n). This means that in calculating the distance to a directed link (j, k), we are really interested in the distance to node j. This actually simplifies the algorithm since the Dijkstra algorithm provides node distances. In SPTA we had to convert these node distances into link distances. In the extended algorithm no such conversion is needed. Shortest hop path length is still used as the metric to judge the validity of link status information.

Another difference in the extended algorithm is in where a node stores information about its adjacent links. This concerns the statement of the algorithm, not the theory behind it. In SPTA, a detected link status change was stored in the port topology table associated with that link. This was reflected in keeping the distance table entry for a port's own link equal to zero. Therefore, a port topology table was always the most reliable source of information about its adjacent link. This port table entry was then placed in the main topology using procedure *compute_topology*. Notice that in Events 2 and 3 of SPTA the link l which had changed was put in set J. Procedure *compute_topology* was then used to install this change in the main topology and to set $C(l) = \text{CHANGE}$.

The above method works well in SPTA, but is inappropriate for the extended algorithm. This can be seen by examining Figure 4.1.2. Assume that node n obtains two successive delay measurements for link (n, m), d_1 and d_2, and sends them to node m. Also assume that node n stores these measurements in its port routing information table R_1. When d_2 becomes available it over writes d_1. Now assume that node m echos d_1 back to node n. If we use SPTA's method described above, this received message would be stored in R_1 and would over write the more recent message, d_2. We can see that a node's port topology table is not a safe place to store delay measurements about the outgoing link associated with that port.

The solution to the above problem is simply to enter delay measurement changes directly into the main topology, as well as storing them in a port topology. Therefore, in Events 2 and 3 of the extended algorithm, set J is not needed. These events must also explicitly send messages about a status change in an adjacent link. Procedure *compute_topology*

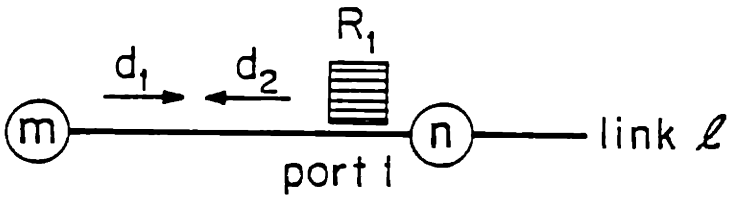

Figure 4.1.2: Updates in the Extended SPTA Algorithm

will not be used to detect these changes. Since a port topology is not being used as the authority on the status of an adjacent link, it is no longer necessary to keep the distance table entry for a port's own link equal to zero. This is reflected in procedure *shortest_path*.

The extended algorithm needs an additional event to handle ordinary delay changes. We call this Event 4. All a node does in this situation is to record the delay change and send a message on each adjacent link.

One final change needs to be made in the way that SPTA computes shortest paths. Since we are considering each link to be two directed links, it is possible for a link to be up in one direction and down in the other. This will only happen for a short time while the algorithm is running. For the purposes of calculating shortest paths, we adopt the convention that if a link is down in either direction in a node's main topology table, then it is considered down.

The statement of the extended SPTA algorithm which follows parallels the statement of SPTA in Chapter 4 nearly line for line. To emphasize that we are considering routing information, we have used a table R in place of the topology table in SPTA. Nearly all of the explanatory comments which were made about SPTA also apply to the extended

algorithm. The algorithm procedures for each node n follow.

Data Structure Definitions: contained at each node n

$R(j,k)$ = node n's main routing information table entry for link (j,k)

$D(j)$ = shortest hop path length to node j

$R_l(j,k)$ = port l's routing information table entry for link (j,k)

$D_l(j)$ = shortest path length to node j that uses link l

$C(j,k)$ = whether $R(j,k)$ has just changed

$L(j,k)$ = the port over which node n believes topology information about link (j,k)
i.e. $R(j,k) = R_{L(j,k)}(j,k)$ for non adjacent links.

J = the set of links about which delay changes have been received

Note: Primed variables refer to information received from neighboring nodes.

Message Format: message from n' to n on link l.

$$U([\overbrace{(j,k)}^{\text{link}}, \underbrace{R_l'(j,k)}_{\text{delay}}], \ldots)$$

This updates port l's topology, for link (j,k), at node n to $R_l'(j,k)$ etc.

Event 1: Initialization: n's node processor is rebooted

Adjacent links are assumed to be DOWN.

$R_i(j,k) :=$ DOWN $\forall$ directed links (j,k) and all ports i

$D_i(j) := \infty \ \forall$ nodes j, and all ports i

$R(j,k) :=$ DOWN $\forall$ directed links j

$D(j) := \begin{cases} 0 & j = n \\ \infty & \forall\ j \neq n \end{cases}$

$L(j,k) :=$ arbitrary, one of node n's ports

end

Event 2: Link l between nodes n and m is now operational with routing measurement r

$J := \{\emptyset\}$

$R(n,m) := R_l(n,m) := r$

execute *shortest_path*(l,n,m)

execute *compute_topology*(l,n,m,J)

send $U([[(n,m),R(n,m)],[(j,k),R(j,k)],\ldots \ \forall$ directed links (j,k) :

$C(j,k) =$ CHANGE) on all links except link l

send $U([[(j,k),R(j,k)],\ldots \ \forall$ directed links) on link l

end

Event 3: Link l between nodes n and m is now non-operational

$J := \{\emptyset\}$

$R(n,m) := R_l(n,m) :=$ DOWN

$D_l(j) := \infty \ \forall$ nodes j

execute *compute_topology*(l,n,m,J)

send $U([[(n,m),R(n,m)],[(j,k),R(j,k)],\ldots \ \forall$ directed links (j,k) :

$C(j,k) =$ CHANGE) on all links

end

Event 4: $U'([[(j',k'),R_l{}'(j',k')],\ldots)$ received on link $l = (n,m)$

$J :=$ {all links in message U'}

$R_l(j,k) := R_l{}'(j,k) \ \forall \ (j,k) : [(j,k) \in J,$ **and** $j \neq n]$

execute $shortest\_path(l, n, m)$

execute $compute\_topology(l, n, m, J)$

send $U([(j,k), R(j,k)], \ldots \ \forall \ (j,k) : C(j,k) = \text{CHANGE}\ )$ on all links

end

Event 5: Routing measurenemt for link $l = (n, m)$ is now r

$R(n,m) := R_l(n,m) := r$

send $U[(n,m), R(n,m)]$ on all links

end

Procedure: $shortest\_path(l, n, m)$

$$d(j) := \begin{cases} 1 & \text{for } R_l(j,k) \neq \text{DOWN and } R_l(k,j) \neq \text{DOWN} \\ \infty & \text{for all other links } (j,k) \end{cases}$$

$$self\_dist(i) := \begin{cases} 0 & \text{for } i = n \\ \infty & \text{for } i \neq n \end{cases}$$

$$neighbor\_dist(i) := \begin{cases} 0 & \text{for } i = m \\ \infty & \text{for } i \neq m \end{cases}$$

$dijkstra(self\_dist, d)$

$dijkstra(neighbor\_dist, d)$

$$D_l(i) := \begin{cases} \infty & \forall \ i : self\_dist(i) < neigh\_dist(i) \\ neigh\_dist(i) + 1 & \text{otherwise} \end{cases}$$

return

Note: The Dijkstra algorithm implementation is the same as for SPTA.

Procedure: $compute\_topology(l, n, m, J)$

$D(i) := \min_k \{D_k(i)\}$ $\forall$ nodes i

for all directed links (j,k) :

$C(j,k) :=$ NO_CHANGE

if $D(j) < \infty$

if $D_l(j) = D(j)$ **and** $(j,k) \in J$

$L(j,k) := l$

else if $D_{L(j,k)}(j) \neq D(j)$

$L(j,k) := \text{argmin}_i\{D_i(j)\}$

if $R(j,k) \neq R_{L(j,k)}(j,k)$

$R(j,k) := R_{L(j,k)}(j,k)$

$C(j,k) :=$ CHANGE

return

End of Algorithm ▮

The structure of this algorithm is nearly identical to SPTA. To analyze, it we can examine the analysis of SPTA and make a few very minor modifications. The time complexity of the extended algorithm is exactly the same in all cases as SPTA. It is still limited by the diameter of the network.

$$T \leq n$$

The communication complexity for k directed link delay changes is the same as SPTA for k status changes. Ignoring the cycling effect gives:

$$C = 2lk.$$

Cycling cannot occur for delay changes since shortest paths do not change. When topology changes occur, the extended algorithm will send twice as many messages as SPTA. This is due to our considering each link to be two directed links. It is certainly possible to differentiate between topology and delay changes to save in communication, but this would greatly complicate the algorithm. One of the most important advantages of the extended algorithm is its simplicity.

The algorithm just presented is an event driven, reliable, and reasonably efficient solution to the routing information problem. In responding to a particular event, it uses the same amount of communication as the ARPANET update procedure, yet it does not require periodic retransmission of information. We have gained efficiency in communication and reliability while only requiring some additional memory and processing. Given the low cost of memory and microprocessors, this seems an excellent tradeoff.

4.2 An Optimal Routing Information Algorithm

In the previous section, we solved the routing information and topology problems together with a single algorithm. Here, we take the opposite approach. We will present an extremely simple routing information algorithm which depends heavily on the existence of a separate topology algorithm. The topology algorithm used must have the resynchronization property discussed earlier. The reason for this approach is to gain efficiency in the solution to the routing information problem. This will undoubtably result in a loss of efficiency in the topology problem solution since we are forced to provide resynchronization. However, since delay changes are far more common than topology changes, this is a desirable trade off.

Consider a network which uses FTA to solve the topology problem. The routing information algorithm will send "delay packets" which are totally separate from topology updates. This routing information algorithm consists of the following simple rules.

1) While FTA is running at a node, the node does not transmit delay measurement messages.
2) When FTA terminates at a node:
    a) the node computes a spanning tree using any appropriate criteria,
    b) the node then sends a message containing the most recent delay measurements for each of its adjacent outgoing links on this spanning tree.
3) When a node detects a delay change on one of its outgoing links, it sends a message containing the new measurement on the spanning tree computed in 2a.

Why does this method work? Recall that the fundamental difficulty in solving the routing information problem is in distinguishing between old and new information. Consider the state of the network when FTA terminates. If a node sends a delay measurement to a neighbor, FTA must have terminated at that neighbor before the message arrives (or a new version must have started). Therefore, the message must be propagated (i.e. rule 1 will not apply). Any delay measurements which were present on the network before FTA started must have arrived before FTA terminates at any node. This is due to FTA's transmitting one message in each direction on each link before beginning the termination phase. Therefore, when new delay measurements are sent on a spanning tree (2b), they cannot be confused with older messages. Since later messages are sent along the same tree, they must arrive at each node after all previous messages. If a delay message is lost, then a topology change must have occurred, and FTA will be started. When FTA completes, each delay measurement will again be updated. We can see that resynchronization leaves the network in a state which makes solving the routing information problem very simple. Essentially, resynchronization creates a time barrier which nodes can use in distinguishing between old and new information.

The only remaining point to address is how a node sends a message along a spanning tree. The most straight forward way would be to include the tree in the delay packets [16]. This would cause the size of the delay packets to become somewhat larger. If this were undesirable, the same spanning tree algorithm could be used at each node. This algorithm would have to construct spanning trees based solely on the network topology. Since each node in a connected set has the same topology when FTA terminates, the same spanning tree would be computed at each node. It would no longer be necessary to include the tree in the delay messages; each node could just relay received messages along the tree.

We have presented the topology and routing information algorithms as completely separate. This separation is not explicitly necessary. For example, the messages sent in 2b could be incorporated into the acknowledgement and termination phases of FTA. The particular choices involved would depend on the network being designed.

Since delay measurements are being sent on a spanning tree, only $n-1$ messages are sent per delay change. The method is optimal in the sense that to inform $n-1$ neighbors of a change, at least $n-1$ messages must be sent. This "optimality" does not extend to the topology problem solution. In addition, when the topology algorithm terminates, a large number of delay measurement packets are sent. This may still be a good trade off since topology changes are much more rare than delay changes. When the topology problem is solved with resynchronization, efficient solutions to the routing information problem are easy to construct.

4.3 Summary and Conclusion

We have presented two algorithms for solving the topology problem. SPTA is very efficient at solving this problem and generalizes into a reasonably efficient solution to the routing information problem. FTA is less efficient at solving the topology problem, but has a resynchronization property which greatly simplifies solving the routing information problem. We have shown how FTA can be used to allow solutions to the routing information problem which send all information on spanning trees and yet are completely reliable.

Each of the algorithms which we have presented is event driven and works by changing existing topology or routing information. These event driven algorithms were shown to provide significant savings in communication, as compared to conventional schemes, without sacrificing reliability.

APPENDIX

Correctness Justification for FTA

In this appendix a set of statements are presented which show that FTA works correctly. FTA is a complicated algorithm, and a detailed proof of correctness is a laborious task. This section should convince the reader that FTA works properly, but lacks the rigor and detail necessary in a formal proof. We wish to show that FTA will work correctly when an arbitrary set of link status changes occurs. FTA is designed to work only under conditions when topology changes are not continuously occurring. To show correctness, we assume that at some time t_s link status changes begin to occur. By some later time t_0 a set S of link status changes has occurred in the network. S is merely the collection of all changes to the variables $s(j)$ contained at each node in the network, and may contain multiple changes for a single link. Time t_0 is followed by a finite interval during which no further status changes take place. We wish to show that if this interval is sufficiently long, FTA will terminate with each of its requirements being met.

We define FTA to be running at a node when that node's update list U contains an entry other than NO_CHANGE. FTA has terminated at a node when it is no longer running. FTA has terminated in a maximum connected set of nodes when it has terminated at each node in the set AND no algorithm messages are present on any link incident on a member of the set. We assume that FTA does not terminate before time t_0. If FTA did terminate prematurely, we could redefine S and t_0 to make the above statement true.

Before t_s, the network is in "steady state." This implies the following about each maximum connected set of nodes, N, in the network.

1) Any two nodes in N have identical topology tables.
2) At each node in N, the topology table entry is correct for each link which is connected to N.

3) At each node N, the topology table entry contains down for each link which is not connected to N.

The steady state assumption is not restrictive since it applies to a single node just after initialization.

Consider an omniscient observer who is aware of the entire steady state topology. Let node n be part of a maximum connected set of nodes N in the steady state before t_s. If the observer made all of the changes in S to the steady state network topology, he would find that n was now connected to a set of nodes N', which may be different from N. We wish to show that a finite time after t_0, node n knows the correct status of each link in N', and considers all other links to be down. We must also show that FTA has terminated in the maximum connected set N'.

We assume in the statements to follow that nodes only receive messages on links for which their DLC status is up. We begin with a simple statement about how FTA reacts to a status change.

Statement 1. *Let the DLC status $s(l)$ of a link l change from up to down at a node. If the node has $T(l) = UP$, then it is either already running FTA with $U(l) = DOWN$ or will immediately start FTA.*

Justification: The statement is true by inspection of Event number 2 of the algorithm. If $T(l) =$ UP, then the test becomes "if $U(l) =$ NO_CHANGE or $U(l) =$ UP." Therefore, if $U(l)$ is not already equal to down, then a new version of FTA is started with $U(l) =$ DOWN. ∎

It is useful in what follows to consider each possible state of $U(j)$ to have a numerical value. Define:

$$U(j) = \begin{cases} 0 & \text{NO\_CHANGE} \\ 1 & \text{UP} \\ 2 & \text{DOWN} \end{cases}$$

It was mentioned earlier that an update list U and a controlling node c uniquely define a version of the algorithm. In what follows, we shall think of each node as keeping the version

number v defined below. Let the network have n nodes and l links. The links are numbered consecutively from 1 to l. Let k be the maximum number of digits needed to represent n. We define the version number of an instance of FTA with update list U and controlling node c to be:

$$v \doteq 10^k \sum_{j=1}^{l} U(j)10^j + c$$

Thus the higher order digits of v contain the state of each link in U, one digit per link. The lower order digits contain the controlling node number c. The version number is not used by the algorithm, but is conceptually useful in showing correctness.

Statement 2. *While FTA is running at a node, that node's value of v is non decreasing and is upper bounded. When FTA has terminated, $v = 0$.*

Justification: A careful examination of the algorithm shows that the value of each entry $U(j)$ in the update list U at a node can only increase, except in procedure *terminate*. The controlling node number c can only be decreased if some entry in U has increased (see Event 5, first and third else statements). Therefore, the value of v must increase since any increase in a $U(j)$ will outweigh a decrease in c. v attains its maximum with every $U(j) = 2$ and $c = n$. After executing procedure *terminate*, a node has $U(j) = 0 \ \forall$ links j and $c =$ NONE (which we take to be zero). Therefore, v has a value of 0. ▌

We will often refer to FTA as running in a set of nodes. The set of nodes in which a version of FTA started by node n will run is defined as: all nodes reachable from node n using links for which ($T(j) =$ UP and $U(j) =$ NO_CHANGE) or $U(j) =$ UP. Examining the algorithm we can see that this is the test used by FTA to determine which links to transmit messages over and expect acknowledgements on. Essentially, FTA uses all working links which are not being turned down. The set of links involved in a version of the algorithm includes all links which are incident on a node which is running the version.

Lemma 1. *If a set of nodes are running FTA with version v and controlling node number c, then node c must be in the set.*

Justification: We use the method of contradiction. Any time a node starts a new version of FTA it makes itself the controlling node. Let n be a node in the above set. n must have been connected to c at some time in order to have received a message identifying version v. If node n became disconnected from c then it must be due to some other link (or links) being turned down. By statement 1, a down update must have been generated about this link. When this update reaches node n, it will no longer be running FTA with version v, which is a contradiction. ∎

Statement 3. *If a node n is running FTA, a finite time later it must execute procedure terminate.*

Justification: Consider all the nodes in the network which are running FTA. Some node c must start a version with a higher value of v than any other node. Node c will be the controlling node for this version, and will broadcast a message identifying it. Any node which receives this message executes *propagate_new_version* and propagates the message. A tree spanning some set of nodes and routed at c will be formed. Acknowledgements will then travel from each node in the set to c. By lemma 1, c must be in the set. The fact that this propagate–acknowledgement scheme works will not be proved. The method is the same as one used in a well known distributed algorithm for generating spanning trees. Node c receives an acknowledgement over each link on which it sent an update message. It then floods an UPDATE_OVER message, and FTA terminates at node c. Some set of nodes will receive this UPDATE_OVER message and also terminate FTA.

Remove the above set of nodes at which FTA terminated from consideration. Some node in the remaining set must have the highest value of v. Apply the preceding arguments to this node, and FTA must terminate in another set of nodes. Continuing this procedure, FTA must terminate at some time at each node in which it was running. ∎

Statement 4. *If the DLC status of a link l at node n changes from down to up and remains up for a sufficiently long time, node n will send BRING_LINK_UP message on link l.*

Justification:

Case 1: When the status change occurs, there are no down updates in progress at n. In this case, BRING_LINK_UP is sent immediately in Event 1.

Case 2: When the status change occurs, there is a down update in progress at n. By statement 3, node n will eventually execute procedure *terminate*. When it does, it will send BRING_LINK_UP on l. ∎

Statement 5. *When FTA is running in a set of nodes, an UPDATE_OVER message must be generated which will be received by each node in the set, and will cause each node to execute procedure terminate.* Note: this does not imply that FTA will terminate since some nodes may transmit BRING_LINK_UP messages when they execute procedure *terminate*.

Justification: From statement 3, a node will at some time execute procedure *terminate*. Therefore, an UPDATE_OVER message corresponding to some version v must have been transmitted by the controlling node, c, of that version. If that UPDATE_OVER message causes each node in the set to execute procedure *terminate* then the statement is true. If it does not, then some node must have a higher version number v' which will be transmitted to each node in the set. Since the version number is bounded, some node in the set must eventually generate a value of v which is never exceeded. When each node in the set receives an UPDATE_OVER message about this version, it can not have a higher version number and must therefore execute procedure terminate. ∎

To analyze FTA it is useful to view its operation as being divided up into several units called phases. The first phase begins when some node in N detects a status change and starts running FTA. Eventually, some node will receive acknowledgements on each of its links and will broadcast an UPDATE_OVER message. If each node that receives this message executes procedure *terminate*, then we consider the phase to end. The phase ends when the last node in the set executes procedure *terminate*. It is important to realize that if an UPDATE_OVER message is being flooded and some node does not execute procedure *terminate* after receiving it, the phase is NOT over. No node can know when a phase

has ended, but an omniscient observer could easily make this determination. The previous statement showed that if a phase of FTA is running in a set of nodes, then the phase must end.

Consider a node which has received an UPDATE_OVER message which eventually results in the end of a phase. We consider the next phase to begin when a node sends BRING_LINK_UP in procedure *terminate* or when another status change takes place. The BRING_LINK_UP message is sent when the previous phase has not resulted in the correct topology, and one or more phases will be required before the algorithm terminates. Notice that by this definition, a new phase can start before an old phase ends. This does not present a problem since we are only concerned with the state of the network at the end of a phase. It must be emphasized that the concept of a phase is only being used to aid in showing correctness, it is not explicitly used by FTA.

Before making use of phases, we need to prove some results about the reconnection of components.

Statement 6. *Let nodes n and n' be in disconnected components and be joined by an inoperative link l. If link l becomes operational, both n and n' must detect that a reconnect is occurring.*

Justification: Consider the two end nodes n and n' of the link l which by being brought up is causing the reconnect. According to the topology at node n, it cannot be connected to n'. Similarly, according to n', it is not connected to n. This must be true since any down updates which caused the two nodes to become disconnected must have ended before n or n' could try to bring link l up. By statement 4, n and n' must transmit BRING_LINK_UP on l. Assume $n > n'$. When node n receives the BRING_LINK_UP message, it will check for reconnects in Event 4 and determine that it is being connected to n'. It will also send an update message on l to n'. When n' receives this update message, which by assumption does not contain any down changes, it will check for reconnects in Event 5 and determine that it is being connected to n. ∎

Statement 7. *Assume some number of components are being reconnected to form a connected set of nodes. The update list at each node in the set must eventually contain an up change for each operating link incident on a member of the set.*

Justification: By statement 6, at least one node in each component must detect a reconnect. Consider a node which detects a reconnect. It will start an update to turn up each link listed as up in its topology. If a link in a component is operating, then each node in the component must list it as up in its topology, or there must be an update in the component to turn it up. This follows from statement 4 since there cannot be any down changes present, or a reconnect would not be occurring. Therefore, each operating link in each component is put into an update list. When these lists are all combined, each node's list contains an up change for each operating link in the set. ▮

Statement 8. *Consider a phase of FTA ending in a set of nodes. When each node executes procedure terminate in response to receiving an UPDATE_OVER message which will end the phase, it installs a new topology. The topologies thus installed at each node are identical.*

Justification:

Case 1: No reconnect has occurred. These topologies were all identical in steady state. We will assume that the statement is true for the previous phase and prove that it is true for the current phase. By induction, it will be true in general. The various nodes in the set may have different topologies since they may have executed procedure *terminate* different numbers of times. If two nodes have a difference in topology for a link then a status change about that link must be in the update list. This is true since one of the two nodes must not have executed procedure *terminate* to remove it. An update about a link only vanishes if each node in the set removes it by executing procedure *terminate*. Therefore, except for links in the update list, the node's topologies are the same. When the update list is installed at each node, the same topology table will be generated.

Case 2: A reconnect has occurred. Since a reconnect is occurring, there cannot be any down changes in the update list. By statement 7, there is an up change in the update list for every working link in the set. Each node's present topology lists as down each

link in other components. Therefore, knowing all the working links uniquely determines the topology. When the update list is installed at each node, the same topology will be generated. ▌

Statement 9. *A finite time after status changes stop occurring, each node in a maximum connected set knows the correct status of each link incident on a member of the set.*

Justification: Assume status changes stop occurring at time t_0. By statement 5, each version of FTA which contained a down change must have ended a finite time after t_0. The only way a down change can be introduced is when a link fails, or when another down change is already present in the update list. Therefore, a finite time after t_0 there are no down updates anywhere in the network, nor will any be generated until the next down status change occurs. At this time, each node in the network knows the correct status of each link which is down. This is true for the following reasons. Consider a down link. If a node is in a different component, from the link then the node considers it down. If the node is in the same component as the link, then by statement 8, the node must have the same topology as at least one of the link's end nodes. Since both end nodes of the link consider it down, so must the node in question.

Now we must consider links which are operating, but are listed as down. When each node executes procedure *terminate* and completes all down changes, it sends BRING-_LINK_UP messages on each working adjacent link which is listed as down in its topology. If a node has not executed procedure terminate since t_0, then by statement 4 it must send BRING_LINK_UP messages. Therefore, an update is started to turn up every working link which is listed as down. These updates will combine and be propagated out to a maximum connected set of nodes. If there are component reconnects, then the updates will contain every working link in a maximum connected set.

Consider a maximum connected set of nodes. The update list discussed above will be formed one phase after the last down change is completed. When this phase terminates, by statement 8, each node in the maximum connected set will have the same topology. We have already shown that this topology is correct for down links. It must also be correct for

working links in the maximum connected set since if it were not, an update would have been started, as shown in the preceding paragraph. Since each node knows the correct status of all links in a maximum connected set, when a node executes procedure *terminate*, it will not send any BRING_LINK_UP messages. Therefore, FTA must terminate. ▌

The last statement shows that FTA solves the topology problem. The fact that it also has the resynchronization property is true by inspection of the algorithm.

REFERENCES

[1] Andrew S. Tanenbaum, *Computer Networks.* Prentice-Hall, Inc., 1981.

[2] Frank Harvey, *Graph Theory.* Addison-Wesley, 1969.

[3] J. M. McQuillan and D. C. Walden, "The ARPA Network Design Decisions," *Computer Networks.* Vol. 1, pp. 243–289, Aug. 1977.

[4] J. M. McQuillan, I. Richer, and E. Rosen, "ARPANET Routing Study – Final Report," Bolt Beranek and Newman Inc. Report No. 3641, Sept. 1977.

[5] Eric C. Rosen, "The Updating Protocol of the ARPANET's New Routing Algorithm: A Case Study in Maintaining Identical Copies of a Changing Distributed Data Base," in Proc. 4^{th} Berkeley Conference on Distributed Data Management and Computer Networks. Aug. 28–30, 1979, pp. 260–274.

[6] J. M. McQuillan, I. Richer, and E. C. Rosen, "The New Routing Algorithm for the ARPANET," *IEEE Transactions on Communications.* Vol. COM–28, No. 5. May 1980.

[7] Radia Perlman, "Fault-Tolerant Broadcast of Routing Information," *Computer Networks.* Vol. 7, Dec. 1983.

[8] E. C. Rosen, "Vulnerabilities of Network Control Protocols: An Example," *Computer Communication Review.* July 1981.

[9] Robert G. Gallager, "Distributed Minimum Hop Algorithms," MIT Laboratory for Information and Decision Systems Report LIDS-P-1175, Jan. 1982.

[10] Dimitri P. Bertsekas, "Dynamic Behavior of Shortest Path Routing Algorithms for Communications Networks," MIT Laboratory for Information and Decision Systems Report LIDS-P-1005, June 1980.

[11] Adrian Segall and Moshe Sidi, "A Failsafe Distributed Protocol for Minimum Delay Routing," *IEEE Transactions on Communications.* Vol. COM–29, No. 5, May 1981.

[12] Steven G. Finn, "Resynch. Procedures and a Fail-Safe Network Protocol," *IEEE Transactions on Communications.* Vol. COM-27, No. 6, June 1979.

[13] James A. Roskind, Edge Display Spanning Trees and Resynchronization in Data Communications Networks, Ph.D. Thesis, MIT, and Laboratory for Information and Decision Systems Report LIDS-TH-1332, Oct. 1983.

[14] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numer. Math.*, vol. 1, pp. 269–271, 1959.

[15] Adrian Segall and Baruch Awerbuch, "A Reliable Broadcast Algorithm," MIT Laboratory for Information and Decision Systems Report, LIDS-P-1177.

[16] J. M. McQuillan, Enhanced Message Addressing Capabilities for Computer Networks," *IEEE Transactions on Communications.* Vol. COM–66, No. 11, Nov. 1978.

Distribution List

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 Copies
Assistant Chief for Technology Office of Naval Research, Code 200 Arlington, Virginia 22217	1 Copy
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 Copies
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, Massachusetts 02210	1 Copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 Copy
Office of Naval Research Branch Office, Pasadena 1030 East Greet Street Pasadena, California 91106	1 Copy
Naval Research Laboratory Technical Information Division, Code 2627 Washington, D.C. 20375	6 Copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20380	1 Copy

Office of Naval Research
Code 455
Arlington, Virginia 22217

1 Copy

Office of Naval Research
Code 458
Arlington, Virginia 22217

1 Copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, California 92152

1 Copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, Maryland 20084

1 Copy

Captain Grace M. Hopper
Naval Data Automation Command
Code 00H
Washington Navy Yard
Washington, DC 20374

1 Copy

Advanced Research Projects Agency
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209

1 Copy

Dr. Stuart L. Brodsky
Office of Naval Research
Code 432
Arlington, Virginia 22217

1 Copy

Prof. Fouad A. Tobagi
Computer Systems Laboratory
Stanford Electronics Laboratories
Department of Electrical Engineering
Stanford University
Stanford, CA 94305