LIDS-P-1702

May 1987

A FAIL-SAFE LAYER FOR DISTRIBUTED NETWORK ALGORITHMS AND CHANGING TOPOLOGIES

Pierre A. Humblet
Stuart R. Soloway

ABSTRACT

We develop a method allowing an algorithm designed for fixed topologies to run on a network with changing topology. The method consists of building a "fail-safe layer" that acts as an interface between the algorithm and the communication network. The communication and time complexities of the method are O(V**2 E) and O(V log V) respectively.

Pierre Humblet is with the Laboratory for Information and Decision Systems, Rm 35-203, Massachusetts Institute of Technology, Cambridge, MA 02139. His work on this research was supported in part by Codex Corporation, by the National Science Foundation under contract ECS 8310698 and by the Army Research Office under grants DAAG29-84-K-0005 and DAAL03-86-K-0171. Stuart Soloway performed this research while he was with Codex Corporation; he is now with Digital Equipment Corporation.

1 INTRODUCTION

A remarkable protocol has been introduced [Fin79] to guarantee reliable end to end data transmission in a network in the presence of arbitrary link and intermediate node failures while not requiring unbounded numbers to identify messages; it also provided a network connectivity test. The basic idea has also been used in [Seg83] to construct other protocols for connectivity test, shortest path and path updating with similar properties.

Although they contain valuable ideas the previous papers share a basic flaw and the algorithms do not always operate correctly. This is demonstrated in a companion paper [Sol 87]. It is possible to modify some of the algorithms to insure the bounded sequence number property, but unfortunately at an increase in running time and communication cost compared to the previous (incorrect) versions. Such a modified algorithm will be explained and proved to be correct in sections 2 and 3, while its complexity will be analyzed in section 4.

Although it is of theoretical importance, the usefulness of achieving the bounded sequence number property for algorithms running in the network layer or above in the ISO/OSI hierarchy should not be overemphasized: the penalty involved in having increasing sequence numbers is often negligible. In addition to the previous family of algorithms which use a single sequence number for each network component, [Per83] and [Hum86] contain topology broadcast algorithms with an unbounded sequence number for each node, while [Spi86] proposes a topology broadcast algorithm that does not rely at all on numbering messages. Since this paper was first circulated, [Awe87] and [Gaf87] have proposed other modifications to Finn's ideas. They have better complexity measures than this one, at the cost of more severe assumptions in the model.

Before describing our method we outline our model. We have a finite network of unreliable links and nodes. Nodes have distinct identities; to simplify the notation we assume that there is at most one link between two nodes, so that a link can be identified by the identities of its end points. Nodes execute distributed algorithms consisting of exchanging messages over links, receiving an external "GO signal" and processing. A distributed algorithm starts at a node on reception of a local "GO signal" or of a message from another node. Message passing is the only way for the nodes to communicate. They have no access to a shared memory or to a global clock.

Regarding the transmission of messages over unreliable links, we assume the existence of a link protocol that interfaces with the processes that execute the algorithms and that has the properties listed below (they are similar but not identical to those in e.g. [Spi86] and [Awe87]). Standard link control procedures, like HDLC with properly chosen timeout values, provide such services (ignoring the possibility of undetected errors). A valid scenario appears in Figure 1.

A link between two nodes X and Y can be either in an Up or Down state at each node independently, subject to the restrictions below. A time interval during which a link is UP at a node is called a Link Up Period (LUP) at that node. Messages can only be sent and received at a node during a LUP there.

- Messages are received in the order they were sent (but some may never be received).

- There is a one to one relationship between some of the LUPs at X and some of the LUPs at Y. A message sent during a LUP can only be received during the corresponding LUP (if any). (A LUP at X may not correspond to any LUP at Y if the link goes UP then DOWN X without changing state at Y.)
- If a message sent during a LUP is never received, all messages sent after it during the same LUP will never be received, and the LUP will be finite.
- The link states at X and Y can remain different only for finite periods of time

Abusing the words slightly, when we say that a link is Up at both ends at some time t, we will mean that t is an element of two corresponding LUPs. Note that it is not possible for a node to determine if a link is currently Up at both ends. However if X sends to Y a message A and receives a message B in answer, then one deduces that the link was Up at both ends between the moments when Y received A and sent B. However X cannot pinpoint a moment where the link was Up at both ends. If Y receives a third message, C, sent by X in answer to B, Y can determine that the link was up at both ends at a precise time in the past.

Similarly nodes can be Up or Down. A node operates without errors while it is Up but loses all its memory when going Down. [Awe87] assumes a Down node maintains its memory. When a node goes Down, all its links go Down within a finite time, and they cannot go back Up while the node is Down. Initially all nodes are Down.

2 A FAIL-SAFE LAYER RESETTING SEQUENCE NUMBERS

2.1 Definition of a fail-safe layer

Assume we have a distributed algorithm designed to operate in a network with static topology and we desire to have it perform correctly in a changing network, i.e. in a way such that if the network topology eventually stabilizes then the algorithm will produce a final output indistinguishable from one it could have produced if it had run once in the final topology.

For some algorithms, like the original Arpanet routing algorithm [McQ77], no modification is necessary. For others, like those to find a spanning tree [Gal83] or a directed spanning tree [Hum83], it is necessary to reinitialize the algorithm whenever a topological change occurs. Naturally the algorithm must be modified so that messages generated during an earlier version are not processed by a later version.

More generally one may think of introducing a protocol layer between the original algorithm and the changing communication network to make the algorithm run correctly, where "correct" is in the sense described above. This layer, the fail-safe layer, can reinitialize the original algorithm, give it a "GO signal", add a "header" to its messages, and pass to it some of the messages received on links. The decision to pass a message is based on the state of the fail-safe layer and not on the message contents, which do not affect the fail-safe layer in any way. ([Cid85] has coined the term "fail-safe compiler", we prefer the word "layer" as there is no need to "process" the original algorithm in any way.)

For example a very simple fail-safe layer implementation keeps track of the largest version number, includes that number in every message, restarts with a larger number on each local topological change or reception of a message with a higher version, and only passes messages corresponding to the largest version. This technique is used e.g. in [Fin79] and [Seg83].

The complexity of a fail-safe layer implementation can be measured by the increase in communication, time or space required by the modified algorithm when compared with those of the original (this will be defined more precisely in section 4). Thus the complexity of a fail-safe layer using restart numbers is unbounded as the size of the numbers increases with the number of restarts.

2.2 A fail-safe layer implementation

In this section we describe a fail-safe layer implementation. However we assume that the nodes have the capability to detect that a special condition described below (inactivity) has taken place. A method to actually implement this detection follows in section 3 while an analysis in section 4 determines the complexity values.

The part of the fail-safe layer that we consider in this section is called the Reset part. It is defined in Figure 2 and we explain its behavior here.

Each node maintains for itself two integers, called LEVEL and SENT_LEVEL, and for each of its links an integer called LINK_LEVEL. To originate a restart, because a local link changes status or for another unspecified reason, a node fail-safe layer acts as in Re_Local incrementing its LEVEL. The increase is noticed by the procedure Re_PROPAGATE which increases SENT_LEVEL and sends a message called New_Restart (abbreviated NR) that carries that level. It will be propagated from node to node, attempting to "capture" the network. LINK_LEVEL keeps track of the last level received over a link. When LINK_LEVEL(L) = LEVEL we say that the link is "Marked" (*).

One can view the diffusion of the NR messages as the expansion of a bubble-gum balloon. Part Re_NR controls how balloons expand. As many events can take place, there can be many balloons in a network and we must specify how they interact. When two balloon walls grow toward each other and they meet, the one with the highest LEVEL wins and continues expanding; if they have the same LEVEL, they simply merge into each other. We will say that a node "restarts" when entering a new balloon, i.e. when it increases LEVEL. When a link is Marked at a node, the opposite end is known to be (or to have been) in the same balloon.

When a balloon stops expanding, either because it has taken over a whole component or because its walls have been absorbed by other balloons, we say that its nodes are inactive (a more precise definition follows) and we allow them to reset LEVEL, SENT_LEVEL and LINK_LEVEL() to 0. If two adjacent nodes were inactive in the same balloon but only one resets and then restarts (with LEVEL = 1) we want the neighbor, which will receive the NR on a marked link, to reset and restart although it has a greater

----

(*) One can also dispense with LINK_LEVEL(), keeping only marked/unmarked information for each link. We do not follow this approach for the sake of clarity.

LEVEL. This is accomplished by the checks on the first line of Re_NR.

A balloon can split in many smaller balloons if links fail. Subballoons of the same original balloon may still be expanding elsewhere in the network while nodes in another part are inactive. Because LEVEL can be reset, parts of the same original balloon may in fact come back through the same node many times; it is thus not obvious that the algorithm terminates !

We now give a precise definition of inactivity.

1) Two nodes X and Y are "joined" at some time if link (X,Y) is Up and Marked at both ends.
2) A "resynch set" is a maximal set of joined nodes.
3) A resynch set is "inactive" if all links adjacent to nodes in the set are Marked at the nodes in the set and if no NR is in transit on an Up link outgoing from a node in the set.
4) A node is inactive if it belongs to an inactive set, else it is active.

In this section we assume that inactivity can sometimes be detected, although not necessarily as soon as it occurs, nor by all nodes at the same time.

2.3 Properties of the fail-safe layer implementation

The first Lemma just points out some properties of the algorithm.

Lemma 1

(a) At a node, LEVEL >= SENT_LEVEL and LEVEL >= LINK_LEVEL(L) for all L in Up_set

(b) When a NR(L) arrives on link Y at a node , LINK_LEVEL(Y) is set to L and the node is or becomes active with a LEVEL not less than L.

(c) A node can only become active because of (1) a local restart, or (2) the reception of a NR() over a link L. At the time the node becomes active, all links are unmarked, except link L in case (2).

Proof: (a) and the first part of (b) follow directly from the statements of the algorithm where LEVEL and LINK_LEVEL() are set. If LEVEL was less than L, or not less than L with LINK_LEVEL(Y) = LEVEL, the node becomes active with LEVEL = L. In the remaining case, i.e. LEVEL not less than L and LINK_LEVEL(Y) less than LEVEL, the node is active by definition.

Part (c) follows from the observation that events at remote nodes cannot make an inactive node become active. The fact that links are marked as stated follows directly from the statements of the algorithm.

****

The second Lemma shows that the first conditions in Re_NR can only be true at inactive nodes and thus that LEVEL can only be reset at inactive nodes.

Lemma 2:

(a) If a node X is active and has link (X,Y) Marked, and (X,Y) is Up at both ends, then Y is active at a LEVEL not lower than X.

(b) If a node X is active, no NR meeting the two conditions on the first line of Re_NR can be in transit from Y to X.

(c) The LEVEL at a node is strictly increasing while the node is active, and so are the LEVELs included in its NR messages.

Proof: (see Figure 3) Part (c) follows directly from part (b), as only the reception of a NR meeting the first two conditions of Re_NR can cause a decrease in LEVEL at an active node.

Assume that the Lemma holds up to some time t, when it fails at node X which is active (at LEVEL L say) and where link (X,Y) is marked. Lemma 1(c) implies that a NR(L) was received from Y since the last time (say v) node X became active (v = 0 if X has always been active). The last such NR was NR(L) that Y sent at time q and that X received at time w, v <= w. Thus:

at X, between w and t, X is active and LEVEL = LINK_LEVEL(Y) = L ______(1)

The Lemma could fail in part (a) or part (b). As Y was active at LEVEL L immediately after q, for part (a) to fail Y must become inactive at some time between q and t (recall that by part (c) this is the only way its LEVEL could decrease). Similarly if part (b) of the Lemma has failed at t there is a NR meeting the first conditions in Re_NR in transit to X. It has been sent by Y at time s > q. The fact that it has a level not exceeding the previous one and this Lemma imply that Y became inactive at some time between q and s.

In either case let r be a time between v and t where Y becomes inactive. Immediately after r, link (X,Y) is marked at Y. Also r is not less than w, as Y cannot be inactive while it has an outgoing NR, thus by (1) link (X,Y) is also marked at X. We conclude that X and Y were joined immediately after time r. This is not compatible with X being active and Y being inactive, thus proving the Lemma.

****

The last Lemma of this section shows that the highest numbered active resynch set will expand:

Lemma 3

If a link (X,Y) is Up at both ends, node X is active at LEVEL L and Y is inactive or it has LINK_LEVEL(X) is less than L, then there is a NR(L) in transit from X to Y.

Proof: Assume the Lemma holds up to time t when it fails. At time s < t, when X last became active at level L, it did send NR(L). Its reception at time v must have caused LINK_LEVEL(X) to be set to L, thus by Lemma 2 Y must have reset at time w, v < w < t after having been active at level L. When Y increased its LEVEL for the last time before w, it sent a NR(L'), L' >= L, to X and this must have been received as Lemma 2 insures that Y was inactive when it reset.
It cannot have been received before s as X would have had to reset sometime between the time of reception and s, and the Lemma would have been violated before t (with the roles of X and Y reversed). However if NR(L') has been received between s and w, L' must be equal to L and the reception will cause LINK_LEVEL(Y) to be set to L. X and Y were joined at time w and one cannot become inactive without the other.
****

Remark that a node can become inactive even though one of its adjacent links is Down at the other end, or carries an incoming NR at a higher level. Such a state of inactivity cannot last. This motivates us to define a resynch set as being "strongly inactive" it it is inactive and if all links adjacent to nodes in the set are Up and Marked at both ends. A node is strongly inactive if it is in a strongly inactive resynch set. External events that cause a node to change from being strongly inactive to being just inactive or active must occur AFTER strong inactivity has been established.

Note that a strongly inactive node is part of a balloon that has never split. Remark also that after a node has become inactive, its resynch set will shrink as some of the other nodes become active. We will define the maximal resynch set of a node as its resynch set at the moment it became inactive. The maximal resynch set of a strongly inactive node is thus an entire network component.

To recap, when inactivity is detected, Re_Inac accomplishes two distinct tasks:
a) it prevents a monotonic level increase by clearing its local variables. Contrary to [Fin79] and [Seg83] this method does not rely at all on sending and tracking differences between restart numbers.
b) it restarts the original algorithm and passes to it messages received on Marked links.

We can now state the key theorem:

Theorem 1

With the fail-safe layer of figure 2, if a finite number of restarts originate, eventually no NR messages are in transit and all nodes in the same connected component end up in the same strongly inactive resynch set.

Proof: Consider a network component the last time a NR originates there; by assumption on the link behavior all links will be Up or Down consistently at both ends. At that time, consider a highest level active node X, with LEVEL = L say. When it last set LEVEL to L, at time t say, it sent NR(L) to all its neighbors and it remains active at least until all have been received.

When the NR(L) arrives at Y it finds Y either

a) inactive or active with LEVEL < L

Lemma 2 insures that LINK_LEVEL(Y) < L and that no other NR(L) is in transit from Y to X. LINK_LEVEL(X) is set to L and a NR(L) is sent to X. X and Y cannot become inactive until the NR(L) arrives at X. When it arrives, both will be joined.

b) active at LEVEL L

Lemma 2 guarantees that LINK_LEVEL(X) is less than L; it is set to L. Y has already sent a NR(L) that has arrived or will arrive at X.

If it has arrived it must have been at or after t, otherwise Lemma 3 would have been violated just before t, when LEVEL at X (and LINK_LEVEL(Y) by Lemma 1) was less than L, no NR(L) was in transit to X but Y was active at LEVEL L.

If the NR(L) arrived at X at t and caused X to become active at Level L, X must have been inactive or active at a LEVEL less than L, and this has been discussed in a) above with the roles of X and Y reversed.

If the NR(L) arrives at X after t, there must have been a time where X and Y each had one NR(L) in transit to the other. Lemma 2 guarantees that LINK_LEVEL(X) and LINK_LEVEL(Y) are < L. When the first NR arrives, at X say, X remains active because it has a NR in transit to Y and Y remains active because LINK_LEVEL(X) < L. When the second NR(L) arrives X and Y will be joined.

Thus all nodes must become active at the highest level and join the same resynch set and no such active node can become inactive unless they all do.

****

2.4 Virtual Networks

The previous theorem says nothing about the behavior of the original algorithm as it is restarted throughout the periods of changes. Before turning our attention to it we introduce the concept of virtual network. A virtual network is a directed graph where the set of nodes and the set of links changes with time, reflecting events occurring during an execution of the fail-safe layer. A virtual network is initially empty and it grows with time as follows. The nth time a node I restarts it becomes a "virtual node" with "virtual identity" (I,n). A directed "virtual link" appears from a virtual node (I,n) to a virtual node (J,m) when (if ever) the NR sent to J by I during its nth restart causes the link to be Marked at J during the mth restart there.

Once a virtual link (or virtual node) appears in a virtual network it never disappears. However when a node I restarts for the n+1th time we will say that virtual node (I,n) "dies". As we have assumed that the nodes have distinct identities, at any time at most one "live" virtual node can have virtual identity (I,n), for some given I and for any n.

The reason why virtual networks are interesting is that if messages are only processed when received on a Marked link, the set of nodes that can influence a distributed algorithm as it executes at (I,n) is the set of nodes that can reach (I,n) in its virtual network (this is so because nodes can only interact by passing messages).

It is important to note that it is possible to have MANY virtual nodes corresponding to the same node in the SAME component of a "virtual network". There can be a virtual link from (I,m) to (J,n), one from (J,n) to (K,o), and one from (K,o) to (I,p), with $p > m$. Consider the following scenario: Initially (I,J) and (J,K) are Up; NR from I to J; NR from J to K during the transmission of which (I,J) goes Down, I becomes inactive and (K,I) goes Up; NR from K to I.

The following properties of a virtual network are easy to establish:

- all the connected nodes in a virtual network have the same LEVEL while they are active.
- If I and J were joined during the mth restart at I and the nth restart at J, then there are virtual links in both directions between (I,m) and (J,n). The converse need not hold; in fact a virtual link may never have been Up at both ends simultaneously.
- To an inactive node maximal resynch set corresponds a subset of a component of the virtual network, virtual links between nodes in the subset come in pairs (one in each direction) and there is at most one virtual node for each node (this follows directly from the definition of inactivity). Such a subset, called the "exchange set", is thus isomorphic to a connected subnetwork of a valid static network (with unique node identities, and possibly "dangling edges" to nodes outside the subset).
- If a node is strongly inactive its exchange set does not have "dangling" adjacent edges to or from outside, and it is isomorphic to a connected component of a valid network. This is the case in particular for the last virtual network component if the topology stabilizes (Theorem 1).

2.5 Behavior of an algorithm used with the fail-safe layer

We now turn our attention to the original algorithm. From the previous discussion one sees that messages received by a node executing a version of the original algorithm must have originated in the exchange set. The design of the fail-safe layer guarantees that the sequence of messages received by the original algorithm on a virtual link is a prefix of the sequence of messages sent over the link. We thus have the Theorem:

Theorem 2

a) The sequences of messages received by the nodes of an inactive resynch set while they execute a version of the original algorithm is a prefix of the sequences of messages that might have been sent by the original algorithm executing in the corresponding static network.
b) Consequently their joint output is a prefix of the joint output that might have been produced by the original algorithm running in a static network
c) After a finite number of topological changes and spontaneous restarts the original algorithm runs in a final network component exactly as in a static network, as its exchange set is isomorphic to the final topology.
****

At this juncture it is interesting to examine what would have happened if the original algorithm had received a "GO signal" at each restart instead of when the node becomes inactive in step Re_Inac. Statement c) in Theorem 2 above would still hold, as in the final component it makes no difference when a message is sent. However statements a) and b) need not hold as the original algorithm could have received messages from nodes outside its exchange set (but still in the virtual network).

Nodes in this larger set need not have distinct identities. Thus the reason why the original algorithm receives the "GO signal" in Re_Inac is not to insure correctness of the fail-safe layer. We will see is section 4 that it is related to its complexity.

The previous theory rests on rather sandy foundations: how is it possible for a node to detect that it is inactive ? In the next section we give and prove the correctness of an algorithm that does detects strong inactivity.

3 DETECTING STRONG INACTIVITY

3.1 An algorithm to detect strong inactivity

It is not obvious at all that strong inactivity should be detectable. Operating in a network with changing topology and where message transmissions times are not bounded how could we ever hope that a node should be able to detect that it belongs to a network component with all links Up at both ends simultaneously ? The answer to the paradox is that we will not detect that a node is strongly inactive NOW, only that it has been (but at a well identifiable point in time, as was the case for the problem of detecting that a link is Up at both ends in section 1).

There are various ways to detect inactivity. The procedure that we will follow is built on an algorithm to find shortest paths in a network. We refer to it has the Shortest Path part of the fail-safe layer implementation. It is based on ideas from [Gal76] and [Hum78]. When run on a network with static topology it allows each node to determine the distance (in hops) to the other nodes in the same network component.

When run as specified in figure 4 it allows a node to determine that it is inactive and when it was strongly inactive. We first give a narrative outline and then prove the main property.

The algorithm is reinitialized at each restart and given a "GO signal" when all Up adjacent links become Marked (thus more often than an original algorithm in section 3) and it runs continuously until it is reinitialized at the next restart (this shortcoming will be repaired later). It only transmits messages on Marked links and only processes messages received on Marked links.

There are two sorts of messages: ID=Node_ID, where Node_ID is a node identity, and DIST=n, where n is an integer. These messages are broadcast by a node on all its adjacent links. A node broadcasts a DIST=n message only after having sent ID=J messages for all nodes J at distances not exceeding n hops.

Messages received on a Marked link are kept in a first-in-first-out queue for that link; we assume that the message at the front of a queue can be read without being removed from the queue.

A node I first sends ID=I, then DIST=0 and it maintains a vector D(). Its Jth entry D(J) is meant to be the minimum distance (in hops) from I to J. Node I also maintains a variable HOP with the property that the D(J) of nodes J at distances not exceeding HOP have been set; it is initially 0.

D(J) is set to HOP+1 and ID=J is sent on all links as soon as the first ID=J message is processed. To insure that the D(J) are set correctly, a node will not read ID=(.) messages from a link on which DIST=HOP has been received until a DIST=HOP message has been received on all links; at that momemt HOP is also incremented and the new value of HOP is sent in a DIST= message.

It is important to note that these distances will be measured in a virtual network (and not in an exchange set) and we should accept the possibility that many nodes with the same identity may be present in a connected component of that network. If at some time no new node identities are found at distance HOP+1 then one might think that all nodes in the connected network component have been discovered and that the radius of the network (as seen by I) is HOP; this would be correct if all nodes had distinct identity (but see figure 5.a where R is set to 1 at 1). When no nodes are discovered at distance HOP + 1, a node merely sets its variable R to HOP (i.e. what it assumes the radius to be) and keeps running the algorithm. If HOP ever exceeds three times R the node must be inactive! The proof that follows will make clear why this is so.

We will consider the operation of the shortest path algorithm at a node I during a time interval between executions of step Sp_Init, i.e. the algorithm as it executes at a virtual node (I,n), assuming that a "dead" virtual node maintains the latest value of the algorithm variables set during its life. (I,n) can only receive a message from a node J if there are virtual links in both directions between nodes (I,n) and (J,m) (for some m), as J (resp. I) will only send (resp. receive) a message on a Marked link. It follows from this that the algorithm behaves as if it was operating in a network with fixed topology containing possibly many nodes with the same ID, and where nodes can stop operating. Where there is no ambiguity we will denote a virtual node (J,k) by J only.

Theorem 3:

1.1 The values of HOP at adjacent nodes can differ by at most one.

1.2 If D(J) is set to a value d different from OO at node I, there is a node J closest to I at distance d from I.

1.3 For all d, if HOP ever reaches d at node I and there is a node J at distance no more than d from I, then D(J) will be set as specified in 1.2.

1.4 If there is a node at distance d from I and all nodes at distance less than d have a unique identity, then R cannot be set to a value less than d.

2. If HOP exceeds 3 R at a node I, then

2.1 I is inactive.

2.2 at the time R was set to HOP, all nodes J with D(J) < OO formed a single network component with all links Up and Marked at both ends (i.e. the resynch set was strongly inactive).

Statements in 1 can be easily verified by induction on HOP, e.g. as in [Hum78]; the reasoning is similar to [Gal76] or [Seg83]).

To prove 2.1, i.e. that HOP exceeds 3 R only at inactive nodes, we consider the moment t (if ever) at the start of Step Sp_Main in the shortest path algorithm in which R is set to the current value of HOP at a virtual node (I,n) (i.e. the node will shortly discover that there is no node with a new identity at distance R + 1 in the virtual network; all links between nodes at distance no more than R + 1 from (I,n) have been Marked). We distinguish between two cases:

A) If there is a dead node (J,s) at distance R or less of (I,n) at time t, we claim that (I,n) can never have HOP › 3 R. Assume to the contrary that it is the first to meet this condition and consider the situation at time t (figure 5.a, I=1, J=3).
By 1.1 HOP at (J,s) is within R at (I,n) of HOP at (I,n), i.e. not greater than 2 R at (I,n); it will never change, insuring that at (I,n) HOP will never exceed 3 R.
B) If all virtual nodes at distance R or less from (I,n) in the virtual network are still alive then
a) the links between those live nodes must still be Marked
b) these nodes may have Marked links to virtual nodes at distance R + 1 from node (I,n) but those virtual nodes (if any) have the same ID as a live node and are thus dead (figure 5.b, I=1, J=3).
Consequently those live nodes constitute an inactive resynch set.

This establishes part 2.1 of theorem 2; we now turn our attention to part 2.2 and show that it held at time t if node HOP exceeds 3 R at (I,n). In light of a) and b) above we only need to show that if HOP exceeds 3 R there cannot be dead nodes at distance R + 1 .

If there is a dead node (J,s) at distance R + 1 from (I,n), there must be another node (J,t) (with t › s) at distance not greater than R from (I,n). Consider the moment when a path of length not exceeding 2 R + 1 first joined (J,t) and (J,s) (this must occur). Some node on such a path had not executed step SP_Start in the shortest path algorithm and thus its neighbor toward (J,s) still had HOP = 0. We can conclude that HOP at (J,s) must have been less than 2 R at (I,n) and this would prevent HOP at (I,n) from exceeding 3 R.

3.2 Optimization of the algorithm

The main contributions of this paper are to identify the correct circumstances where LEVEL can be reset (i.e. inactivity), introduce the concept of virtual network and show how to detect strong inactivity without requiring Down nodes to maintain their memory. We end this section on a more routine note: optimizing the algorithm somewhat with an eye to the complexity measures that will be introduced in the next section.

There are two main problems: first LEVEL can be reset, but this does not guarantee that it remains bounded (consider the situation described in figure 6). Second the Shortest path algorithm runs "forever".

Building on a remark of [Fin79], boundedness of LEVEL can be insured by not allowing links to come Up until all nodes in a resynch set have reset LEVEL; the only way LEVEL could keep increasing is because links fail, and thus it will remain bounded. To implement the idea we will not allow LEVELs at adjacent nodes to differ by more than 1, insuring that all nodes in a network component are active if LEVEL at a node equals the number of nodes. We will also be careful about not letting links come Up until all nodes have reset. To this end we will rely on the following observations:

A node has detected that it is inactive when HOP $>$ 3 R. Lemma 4 below shows that if HOP $>$ 7 R at some node in a resynch set, then HOP $>$ 3 R at all nodes. Similarly if HOP $>$ 15 R at a node, HOP is greater than 7 R at all nodes. Talking figuratively, a node that has HOP $>$ 15 R knows that all nodes know that all nodes know that they are inactive ! (This method of detecting the status of other nodes is inefficient, but it avoids introducing new types of messages).

To implement these ideas we will modify the Reset and Shortest Path parts of the fail-safe layer as specified in the Appendix. We will only discuss here the essential new features of the algorithm.

a) The Reset part will only allow one NR in transit on a link by delaying the sending of NR(LEVEL) while SENT_LEVEL > LINK_LEVEL. This does not affect the validity of the algorithm, as the situation could have happened before if the messages that caused the increase of LEVEL past SENT_LEVEL had been delayed by the network.

b) We delay including in Up_set a link that just came Up, keeping it in Wait_set instead. The failure of a link in Wait_set does not trigger a restart. To insure that LEVEL remains bounded it is enough to wait until HOP > 7R (this will be called the 7R option) to move all links from Wait_set to Up_set, and to send a WAKE message on them to invite the remote nodes to initiate a resynch if they already have HOP > 7R (i.e. when all nodes have reset LEVEL). If we receive a NR on a link in Wait_set we reply with a WAIT message; this will cause the link to be placed back into Wait_set at the remote end.

For reasons that will be made clear in section 4, we actually wait until HOP > 15 R to move links from Wait_set to Up_set. We also make another change: on receiving an NR on a link in Wait_set, if the NR comes from a node J with D(J) not OO, (i.e. we believe that the node is in the same resynch set) and HOP > 7 R, then we place the link in Up_set and we process the NR, instead of issuing a WAIT.

Note that in both the 7R option and in the code described in the Appendix LEVEL is still reset when HOP › 3R.

To solve the second problem (the Shortest path algorithm running forever) we simply modify the shortest path algorithm to stop processing messages when they are not needed, i.e. when HOP is greater than 15 R. To insure that neighboring nodes, whose Rs differ by at most 1, can also reach the point where HOP › 15 R, the node will send 15 DIST= messages before stopping (again we use this method for its simplicity).

The main results of the modifications are captured in 3 Lemmas where V denotes the number of nodes and E the number of links.

Lemma 4: For 0 ‹ N ‹= 7, if HOP › (2 N + 1) R at I then HOP › N R at all nodes J in I's resynch set. This Lemma follows from the facts that R at J ‹= 2 R at I and that HOP at I and HOP at J differ by at most R at I.
****

Lemma 5: Between the moments when (1) a node is in a strongly inactive resynch set where all nodes have reset LEVEL and (2) the node next becomes strongly inactive:
a) LEVEL cannot exceed SENT_LEVEL + 1 at the node
b) SENT_LEVEL - M at nodes M hops away cannot exceed SENT_LEVEL at the node, for all M (defining a hop as a link that is Up and in Up_set at both ends).

Proof: Assume the theorem does not hold for the first time at some node I.
It cannot be because part a) failed. The increase in LEVEL cannot be

due to a local restart, as in that case LEVEL is set to MAX(LEVEL,SENT_LEVEL+1). It cannot be due to the reception of a NR on some link (I,J) either because of part b).
Thus it must be because part b) failed. Consider M = 1 first, with SENT_LEVEL at I larger than SENT_LEVEL - 1 at a neighbor J. It cannot be because link (I,J) just came up, as it will be included in Up_set only when SENT_LEVEL is 0. It cannot be because node J has reset its SENT_LEVEL, as node J cannot reset until I becomes inactive. It cannot be because SENT_LEVEL increased too much at I: SENT_LEVEL at I cannot increase if it is more than LINK_LEVEL(J), which does not exceed SENT_LEVEL at J (because J has not reset) and part a) guarantees that any increase is at most one.

The proof continues with an induction on M that we omit for the sake of brevity.
****

Lemma 6: LEVEL cannot exceed V + 2 E.

Pf: The previous Lemma shows that if LEVEL reaches V at a node I then either I has become strongly inactive since the last time it was in a strongly inactive set where all nodes have reset LEVEL, or LEVEL is greater than 0 at all connected nodes. In either case all connected nodes have had 7 HOP <= R and none will add a link in Up_set until all connected nodes form a strongly inactive resynch set with LEVEL reset at all nodes (i.e. HOP > 7 R at some node). LEVEL can only increase because links included in some Up_set go Down.
****

4 COMPLEXITY ANALYSIS

In this section we examine the space, communication and time complexities of our fail-safe layer. We assume that the algorithm originates K times in a network with V nodes and E links (with E › V for simplicity).

Space Complexity:

Space complexity is defined as the amount of memory (in bits) that is required to hold the variables used by the fail-safe layer at each node. One sees easily that at most O(V log(E)) bits are required. This can be reduced to O(V) by not keeping track of LINK_LEVEL().

Communication Complexity

Denote by C(K,V,E) the maximum number of messages (of length O(log(V))) that need to be exchanged by the original algorithm and the fail safe layer if K originating events take place. Denote by Cs(V,E) the maximum number of messages exchanged by the original algorithm operating in a fixed topology of V nodes and E links. The communication complexity of the fail-safe layer is then defined as Cf(V,E) = sup (over K) C(K,V,E) / K - Cs(V,E). This definition follows [Awe87].

Cf can be computed as follows: between successive instants when a node becomes strongly inactive its LEVEL can only increase, and increases past V must result from a link failure. Thus the number of restarts per originating event is no more than V.

At each restart the Reset part generates at each node no more than O(V) messages on each link. The sum over all links of such messages cannot exceed O(V E) per restart, or O((V**2) E) per originating event. The original algorithm is reinitialized only when a node becomes strongly inactive, thus it executes at most K times in a network isomorphic to a static network and its total communication complexity is at most K Cs(V,E). We conclude Cf = O(V**2 E).

Note that if the original algorithm is reinitialized at each restart (as briefly considered in section 2) it may have to run O(K V) times. Moreover its communication complexity is not necessarily bounded by Cs(V,E) as it may operate in a network where nodes do not have distinct identities. It becomes impossible to bound Cf by a function that depends only on K, V and E.

One might also wonder how many times a node can restart due to a single originating event (i.e. how many times a balloon can include the same node). The answer is O(K) times; it is surprising in light of the previous complexity result.

Time complexity:

The time analysis is somewhat peculiar, as no clock is assumed and message transmissions and the notifications of link failing or coming Up are only required to take finite time. For the purpose of comparing the running times of algorithms in an environment where message transmission times dominate and are fairly linear with message length we will assume that transmitting a message (of O(log(V)) bits) takes unit time, and that processing is instantaneous. We will further assume that any messages longer than O(V) bits (if any in the original algorithm) are divided in packets of O(V) bits and that we maintain a first_in_first_out queue of packets waiting to be transmitted. Whenever Re_PROPAGATE issues a new set of NR messages, all packets are flushed from the queue (they are obsolete anyway).

The time complexity of the fail-safe compiler is defined (following [Gaf87]) as the time elapsed between the moment of the last notification of a topological change and that when the original algorithm terminates for the last time, minus the time complexity of the original algorithm.

After the last origination takes place (at time 0 say), the highest LEVELs propagate through the network; this can take time O(V). The shortest path algorithm then completes in time O(V), at the latest by some time T = O(V).
The original algorithm starts by time T if there are no links in Wait_sets, but the situation is more complicated when there are such links. By time T all have carried WAKE messages in both directions.

Under the 7R option, if there are links in Wait_set at least one link will eventually be in Up_set at both ends, and a Resynch will start. The subsequent NR receptions might cause WAIT messages to be issued, keeping all other links in Wait_set, so that the running time of the algorithm might be as high as O(E T).

The situation is different for the 15R option described in the Appendix, as it insures that all links in Wait_set WHOSE END NODES ARE IN THE SAME RESYNCH SET will be moved to Up_set and stay there. This is so because a NR sent from one end must find the other end with HOP > 7 R, causing it to be moved to Up_set (if it not there already), instead of generating a WAIT reply. The shortest path algorithm terminates by time

2T, at that time links in Wait_set connect different resynch sets.
If at time 2T an inactive resynch set only has links in Wait_set going to other resynch sets, then a restart must have started by time 3T and propagate normally on at least one such link.
We can conclude that by time 4 T all resynch sets that still have links in Wait_set will have at least two nodes. Repeating the argument by time 4 N T all resynch sets with links in Wait_sets have at least 2**N nodes. Thus in time O(V log V) the original algorithm must have started for the last time at all nodes and that the time complexity of our compiler is O(V log V).

The time complexity remains the same in a model where message transmissions take unit time, no matter their lengths. This measure is appropriate when there is much overhead associated with the transmission of a message.

REFERENCES

B. Awerbuch, "Fail-Safe Compilation of Protocols on Dynamic Communication Networks", MIT-LCS, 1987

I. Cidon and R. Rom, "Protocol Extensions", Technical Report, Technion, Haifa, November 1985

S.G. Finn, "Resynch Procedures and a Fail-Safe Network Protocol", IEEE Trans. Commun., vol. COM-27, pp. 840-845, June 1979.

E. Gafni and A. ?,"Topology Resynchronization: A New Paradigm for Fault Tolorant Distributed Algorithms",UCLA Technical Report 1987.

R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees", ACM Trans. Program. Lang. Syst., vol. 5, pp. 66-77, Jan. 1983.

P.A. Humblet, "A Distributed Shortest Path Algorithm", Proceedings of the International Telemetering Conference, ITC'78, Los Angeles, CA, November 1978

P.A. Humblet, "A Distributed Algorithm for Minimum Weight Directed Spanning Trees", IEEE. Trans. Commun., vol. COM-31, pp. 756-762, June 1983.

P.A. Humblet, S.R. Soloway and B. Steinka, "Algorithms for Data Communication Networks - Part 2", Submitted for publication, 1986.

J.M. McQuillan and D.C. Walden, "The ARPANET design decisions", Comput. Networks, vol 1, Aug. 1977.

R. Perlman, "Fault-Tolerant Broadcast of Routing Information", Proc. IEEE Infocom '83, San Diego, 1983.

A. Segall, "Distributed Network Protocols", IEEE Trans. on Info. Theory, Vol. IT-29, no. 1, Jan. 1983.

J. Spinelli, "Broadcasting Topology and Routing Information in Computer

Networks", submitted for publication.

APPENDIX: The final fail-safe layer implementation

THE RESET PART AT NODE I

```
Re_Local On receiving a local signal {
If (signal = node coming Up) { LEVEL = SENT_LEVEL = 0
                               Up_set = Wait_set = {}
                               call SP_INIT() }
If (signal = link L coming Up) {LINK_LEVEL(L) = 0
                               if ( HOP › 15 R) {
                                  Up_set = Up_set U {L}
                                  send WAKE on link L }
                               else { Wait_set = Wait_set U {L} }
                               }
If (signal = link L coming Down) {If ( L is in Up_set) {
                                  Up_set = Up_set \ {L}
                                  LEVEL = MAX(LEVEL,SENT_LEVEL+1) }
                               If ( L is in Wait_set) {
                                  Wait_set = Wait_set \ {L} }
                               }
If (signal = restart for another reason) { LEVEL = MAX(LEVEL,SENT_LEVEL+1) }

call Re_PROPAGATE()
}

procedure Re_PROPAGATE(){
if ((LEVEL › SENT_LEVEL) AND
    (SENT_LEVEL <= LINK_LEVEL(K) for all K in Up_set ) {
   SENT_LEVEL = LEVEL
   send NR(LEVEL) on all links in Up_set
   reinitialize the original algorithm
   call SP_Init() }
}

procedure Re_Awake(){      ! Awake all links in Wait_set
       Send WAKE on all links in Wait_set
       Up_set = Up_set U Wait_set
       Wait_set = {}
}

Re_Wait) On receiving WAIT on link L : {
Up_set = Up_set \ {L}
Wait_set = Wait_set U {L}
if (LEVEL = LINK_LEVEL(K) for all K in Up_set) call Sp_Start() }

Re_Wake) On receiving WAKE on link L : {
if (HOP › 15 R) {
       LEVEL = MAX(LEVEL,SENT_LEVEL+1)
       call Re_PROPAGATE()
       }
}

Re_NR) On receiving NR(NEW_LEVEL) on  link  L {
if (L is in Wait_set) {
   if ((HOP <= 7 R) OR (D(L) = 00)) { send WAIT on link L }
   else {
      Up_set = Up_set U {L}
      Wait_set = Wait_set \ {L}  }
   }
if (L is in Up_set) {
   if ((LEVEL = LINK_LEVEL(L)) AND (NEW_LEVEL <= LINK_LEVEL(L))) {
      LEVEL = SENT_LEVEL = LINK_LEVEL(K) = 0 for all K in Up_set }
   LINK_LEVEL(L) = NEW_LEVEL
   LEVEL = MAX(LEVEL,NEW_LEVEL)
   call Re_PROPAGATE()
   if (LEVEL = LINK_LEVEL(K) for all K in Up_set) call Sp_Start()
}

Procedure Re_Inac() {
   LEVEL = SENT_LEVEL = LINK_LEVEL(K) = 0 for all K in Up_set
   give a "GO signal" to the original algorithm }

Re_Pass) On receiving on link L a message for the original algorithm {
   if (LINK_LEVEL(L) = LEVEL) pass it to the original algorithm }
```

SHORTEST PATH PART AT NODE I

```
Procedure Sp_Init() {
R = D(J) = 00, V J
HOP = D(I) = 0,
Flush the queues for all links
If (Up_set = {}) {
       HOP = 00
       R = 0
       call Re_Inac() }
}

Procedure Sp_Start() {
Send ID=I and DIST=0 on all links in Up_set}

Sp_Main) When receiving a ID= or DIST= message on a Marked link L
       with HOP <= 15 R {
Place it in the queue for L
Loop:
   If (there is some J with an ID=J message at the front of a queue){
      dequeue the message
      if D(J) = 00, {
         D(J) = HOP + 1
         send ID=J on all links in Up_set }
      goto Loop
      }
   If (there is DIST=HOP at the front of ALL queues J, J in Up_set) {
      dequeue one message from all queues
      If (R = 00 and ({nodes J | D(J) = HOP+1 } = {}) { R = HOP }
      HOP = HOP + 1
      Send DIST=HOP on all links in Up_set
      If (HOP = 3 R + 1) { call Re_Inac() }
      If (HOP = 15 R + 1) {
         call Re_Awake()
```

```
           send DIST=HOP+1, DIST=HOP+2,.. DIST=HOP+15 on all links in Up_set}
        goto Loop
        }
    }
```

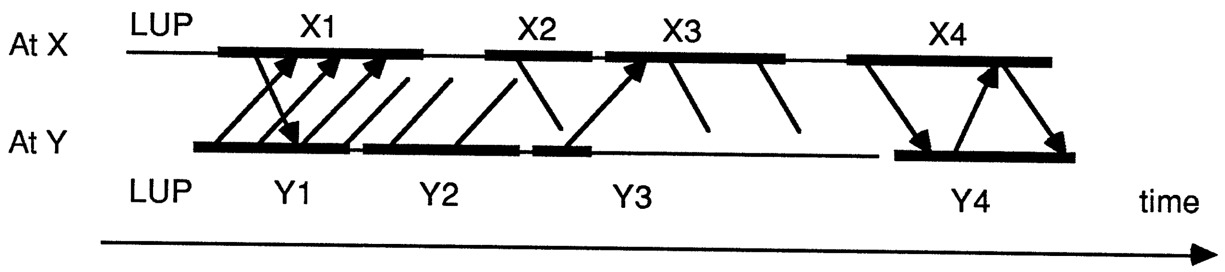

Figure 1

▬ Link Up Period

— Link Down

↗ Successful transmission

／ Unsuccessful transmission

LUP's X1, X3 and X4 correspond to Y1,Y2 and Y4 respectively.
LUP's X2 and Y2 do not correspond to any other LUP
Note that the link was never down at both end between Y1 and Y2, and that the corresponding LUP's X3 and Y3 do not overlap in time.

Figure 2: The Reset part of the fail-safe layer

```
Re_Local Originating a new restart on a local signal {
If (signal = node coming Up) {
   LEVEL = SENT_LEVEL = 0
   Up_set = {} }
If (signal = link L coming Up) {
   Up_set = Up_set U {L}
   LINK_LEVEL(L) = 0
   LEVEL = MAX(SENT_LEVEL+1,LEVEL) }
If (signal = link L coming Down) {
   Up_set = Up_set \ {L}
   LEVEL = MAX(SENT_LEVEL+1,LEVEL) }
If (signal = restart for another reason) {
   LEVEL = MAX(SENT_LEVEL+1,LEVEL) }

call Re_PROPAGATE()
}

procedure Re_PROPAGATE (){
if (LEVEL > SENT_LEVEL ) then {
   SENT_LEVEL = LEVEL
   send NR(LEVEL) on all links in Up_set
   reinitialize the embedded algorithm  }
}

Re_NR) On receiving NR(NEW_LEVEL) on  link  L {
if ((LEVEL = LINK_LEVEL(L)) AND (NEW_LEVEL <= LINK_LEVEL(L))) then {
      LEVEL = SENT_LEVEL = LINK_LEVEL(K) = 0 for all links K in Up_set }
LEVEL = MAX(LEVEL,NEW_LEVEL)
LINK_LEVEL(L) = NEW_LEVEL
call Re_PROPAGATE{}
}

Re_Inac) On a node  detecting inactivity {
   LEVEL = SENT_LEVEL = LINK_LEVEL(K) = 0 for all links K in Up_set
   Give a "GO signal" to the embedded algorithm }

Re_Pass) On receiving on link L a message for the embedded algorithm {
   if (LINK_LEVEL(L) = LEVEL) pass it to the embedded algorithm }
```

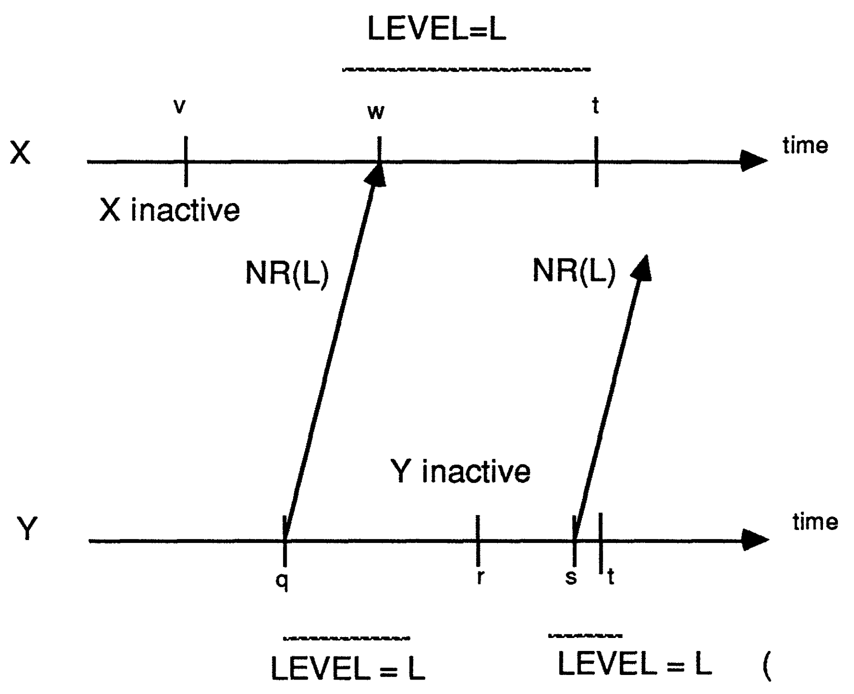

Figure 3: illustration for Lemma 2

Figure 4:
Shortest Path part of the fail-safe layer at node I:

```
Sp_Init) whenever procedure Re_PROPAGATE() finds LEVEL › SENT_LEVEL:{
R = D(J) = OO, V J # I
HOP = D(I) = O,
Flush the queues for all links
}

Sp_Start) When all Up adjacent links become Marked {
Send ID=I and DIST=O on all Links in Up_set }

Sp_Main) When receiving a ID= or DIST= message on a Marked link L {
Place it in the queue for L
Loop:
      If (there is a J with an ID=J message at the front of a queue){
              dequeue the message
              if D(J) = OO, {
                      D(J) = HOP + 1
                      send ID=J on all links in Up_set }
              goto Loop
              }
      if (there is DIST=HOP at the front of ALL queues J, J in Up_set) {
              dequeue one message from each queue
              If (R = OO and ({nodes J | D(J) = HOP+1 } = {})
                      then { R = HOP }
              HOP = HOP + 1
              Send DIST=HOP on all links in Up_set
              If (HOP = 3 R + 1) signal inactivity
              goto Loop
              }
}
```

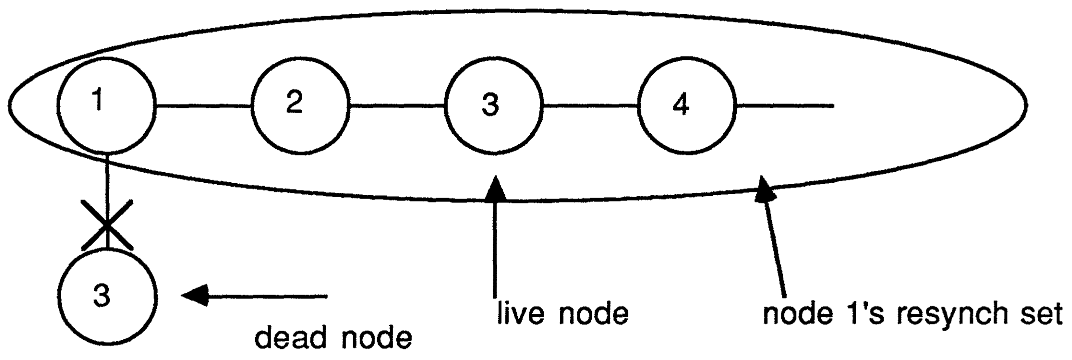

Figure 5 a

Node 3 appears twice in the virtual network,
R(1) is set to 1
NR messages may still be in transit
to the right of node 4 so that the inactivity
of node 1 cannot be guaranteed in this
virtual network.

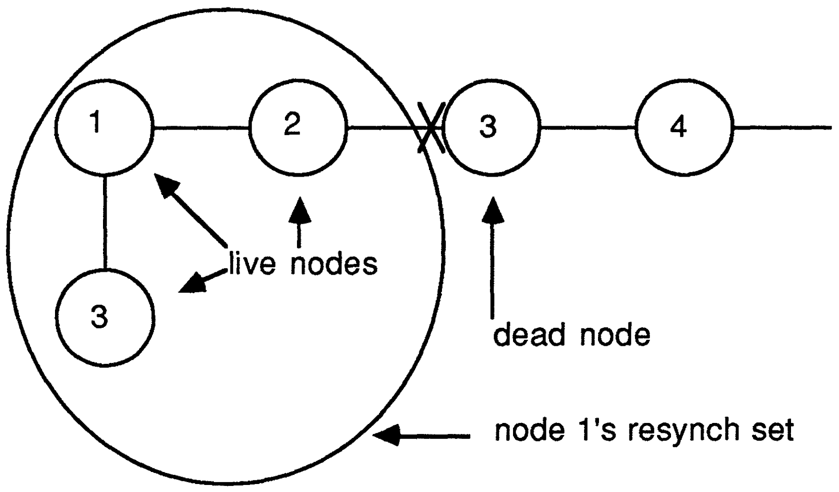

Figure 5 b

Node 3 appears twice in the virtual network,
R(1) is set to 1
nodes 1,2 and 3 may become inactive,
but not strongly inactive. HOP will not exceed 3R.

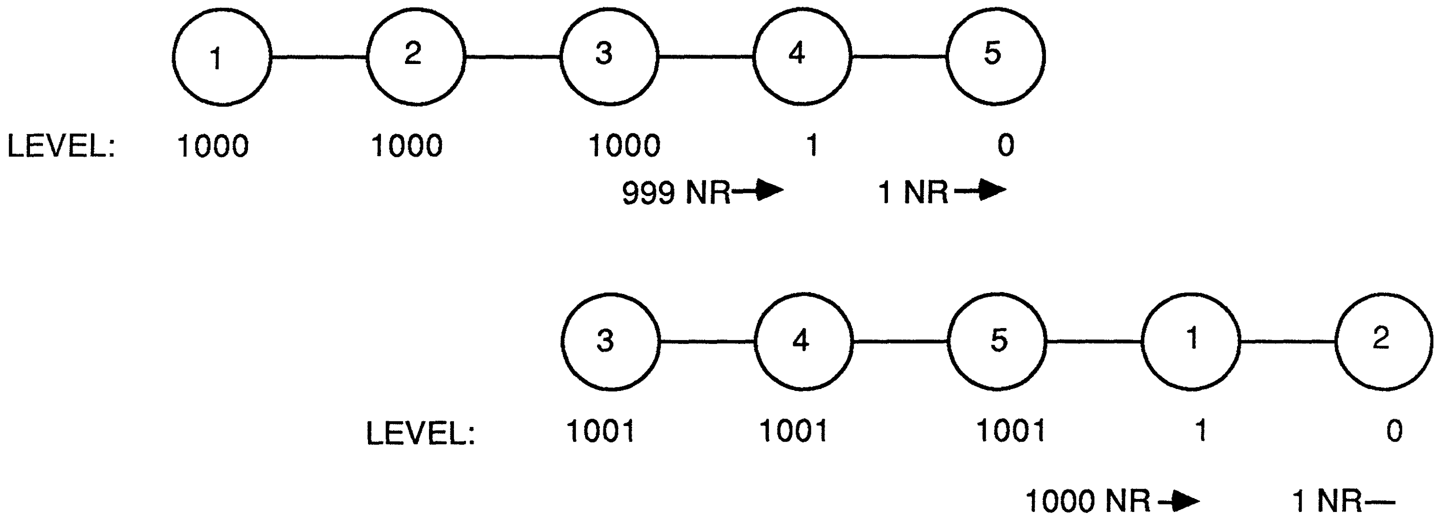

Figure 6

Example where LEVEL increases without bound because topological changes occur faster than NR's travel.
The two leftmost nodes disconnect from the other nodes, become inactive, then connect with the rightmost node, while the 999 NRs in transit between the middle nodes are forwarded to the right. Subsequently 1000 NRs are in transit between middle nodes. Repeating this scenario ....