May 1986 LIDS-P-1564

ON DISTRIBUTED NETWORK PROTOCOLS FOR CHANGING TOPOLOGIES

Stuart R. Soloway
Pierre A. Humblet

ABSTRACT

A number of distributed network protocols for reliable data transmission, connectivity test, shortest path and topology broadcast have been proposed with claims that they operate correctly in the face of changing topology, without need for unbounded numbers to identify different runs of the algorithms. This paper shows that they do not possess all the claimed properties. However some of them can be modified so that that their correct operation can be demonstrated, at a cost of longer running time and of higher communication complexity.

Stuart Soloway is with Codex Corporation, 20 Cabot Boulevard, Mansfield MA 02048

Pierre Humblet is with the Laboratory for Information and Decision Systems, Rm 35-203, Massachussets Institute of Technology, Cambridge, MA 02139. His work on this research was supported in part by Codex Corporation and in part by the National Science Foundation under contract ECS 8310698.

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. These works relied partially on techniques set forth in [Gal76].

This article shows that although they contain valuable ideas the previous papers share a basic flaw and that the algorithms do not always operate correctly. This will be demonstrated in the case of [Fin79] in the following section. 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 3 and 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, as the overhead 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.

Before proceeding with Finn's algorithm we outline our model. We have a finite network of unreliable links and nodes; 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 that consist of exchanging messages over links and processing.

We assume the existence of a link protocol that provides the following interface to the processes that execute the algorithms (a valid scenario appears in Figure 1):
A link between two nodes X and Y can appear to be either Up or Down at each node independently, subject to the restrictions below. Messages can only be sent and received at a node while the link is Up there.
If a link goes Down at X while it is Up at Y, then it will go Down at Y within a finite time.
If a message is transmitted during the mth Link Up Period (LUP) at X then
- either it is never received at Y; in that case the link is declared Down at X within a finite time.
- or it is received correctly within a finite time, during the nth LUP at Y say. In that case no message sent after it will be received before it, and, for all k, the kth message received during the nth LUP at Y (resp. mth LUP at X) is the kth message sent during the mth LUP at X (resp. nth LUP at Y).

Similarly nodes can be Up or Down. A node operates without errors while it is Up but loses all its memory when going Down. When a node goes Down, all Up links at adjacent nodes go Down within a finite time.

2 FINN'S ALGORITHM

This section outlines the basic mechanism of Finn's algorithm and shows the problem that can appear in presence of link or node failures. We view the algorithm as only providing for a connectivity test, i.e. when it halts at a node after a finite number of link or node failures, the node is aware of what other nodes are in the same connected network component.

Each node I will maintain a vector D(I) with an entry D(I)(J) for each node J in the network. D(I)(J) can take the values 0,1 and 2 with the following meanings.
A value of 0 indicates that it is not known at node I if J is in

the same connected component.
The value 1 indicates that J is in the same component, but that the identities of all its connected neighbors might not be known as no message has been received from all of them.
The value 2 indicates not only that J is in the same component, but also that a message has been received from all its connected neighbors (thus D(J)(K) is 1 or 2 for all connected neighbors K of J).

Initially D(I) is set to all 0, except D(I)(I) which is set to 1 (at all nodes I). Nodes exchange their identities and D(.) vectors with their neighbors; when a vector D(K) is received a node I, D(I)(J) is set to MAX(D(I)(J), D(K)(J)) for all J and if D(I)(L) is equal to 1 or 2 for all neighbors L of I then D(I)(I) is set to 2. If this update causes any change in D(I) the updated value of D(I) is communicated to all neighbors of I, where similar updates take place.

It is easy to see that in case of a "cold start" in absence of topological change the algorithm will terminate a node I with the entries of D(I) set to 0 or 2, the later values corresponding to nodes in the same component as I.

The case of changing topology can be handled quite naturally by restarting the algorithm every time a topological change is noticed. To distinguish algorithm cycles it is enough to use restart numbers, choosing a larger number at each restart. By including the restart number in each message one can insure that all nodes in a connected component participate in the latest restart (discarding messages from previous ones). The problem with this approach is that restart numbers increase monotonically.

To remedy this problem [Fin79] has suggested that a node transmit only the difference between its current restart number and the previous one, and that each node maintain "link counters" to track the differences between the numbers of the restarts taking place at its neighbors. Transmitting and tracking differences solves the problem of monotonic increasing sequence numbers, but poses a problem when a link comes up: with respect to what should the difference be interpreted ? To solve this last problem [Fin79] delayed the processing of a link coming UP until both ends have terminated the algorithm and all "link counters" are zero; the "link counter" of a link coming up in these conditions is initialized to zero. (We refer the reader to [Fin79] for the details).

To see that this does not work consider the following example where there are 4 nodes.

```
 1 -------2--------3          4
```

Initially links (1,2) and (2,3) are Up, no node has started the algorithm, all link counters are 0.
Node 3 starts its first restart and transmits D(3) = (0,0,1,0) to 2.
In answer node 2 transmits D(2) = (0,1,1,0) to 1 and 3
Node 3 replies by sending (0,1,2,0), that message arrives at 2 and is forwarded to 1.
At this point the link between 2 and 3 fails, but it takes at very long time for the failure to be noticed at 2. During that time node 3 terminates the algorithm then connects with node 4 and both run the algorithm until completion. When this is done the link between 1 and 3 can come Up, as node 1 has not yet joined any restart. We then have the following picture of the network:

```
1------2----      3 -------4
|                 |
|-----------------
```

Both 1 and 3 start the algorithm by sending (1,0,0,0) and (0,0,1,0) to their respective neighbors 2,3 and 1,4; assume that the message from 3 to 4 suffers a long delay.

Now node 1 receives (0,1,1,0) from 2 and (0,0,1,0) from node 3; It sends (2,1,1,0) to its neighbors 2 and 3.
After receiving this message node 2 has a vector (2,2,2,0), it sends it to 1 and terminates the algorithm, even though it does not know about 4! (in fact node 4 has not even started the

algorithm in its current network component).

At this point the algorithm has halted at 2 without fulfilling its promise, but one might hope that this is not disastrous: node 2 will eventually receive notification that its link to 3 has failed and will restart and, in absence of topological changes, correctly terminate. However another event with catastrophic consequences can also occur.

It is now acceptable for link (4,2) to come Up, as none of its extremities are involved in the algorithm. The situation is then as follows

```
       -------------------
       |                 |
1------2----    3 -------4
|               |
-----------------
```

Nodes 2 and 4 restart (the second time for 2, but only the first for 4 in the current network component) indicating a restart number increment of 1. The restart from 2 will be interpreted by 1 as being the SECOND one; node 1 will immediately also restart, answer to 2 and notify 3. The restart from 4 will be interpreted by 3 as being the FIRST one, and when notice of a second restart arrives from 1 node 3 dutifully relays it to 4, where it will arrive after the first restart from 1, thus triggering a message to 2 that a new restart is to take place. Node 2 then notifies 1 that a new restart (the THIRD one !) is to occur and the reader realizes that the algorithm is now chasing its tail, never terminating. That node 2 is eventually notified that its link to 3 has failed does not help.

A similar counterexample can be constructed for the algorithm EMH-Version B in [Seg83]. The proof of theorem EMH-B-1 has a flaw in the second column of page 32.

3 A NUMBERING ALGORITHM

In this section we give an algorithm that allows all nodes in a network to identify the restarts while not requiring monotonically increasing numbers, and we prove its correctness.

However in order to achieve that goal we assume that the nodes have the capability to detect that a special condition (inactivity) has taken place. A method to actually implement this detection follows in section 4.

We now proceed with the description of the numbering algorithm. Each node maintains for itself an integer, called LEVEL, and for each of its links a binary flag. Setting (resetting) the flag associated with a link is called marking (unmarking) the link. To signal the beginning of a restart, nodes exchange messages called New_Restart (abbreviated NR) that carry a level.

The numbering algorithm is defined as follows:

A) For any reason a node can originate a restart, but it must do so when a local link is detected as changing status (going Up or Down):

```
-Increment LEVEL
-Send NR(LEVEL) on all adjacent Up links
-Unmark all adjacent links
```

B) When receiving NR(NUMBER) on link L, a node acts as follows:

```
-If NUMBER > LEVEL or link L is marked:
    Set LEVEL to NUMBER
    Send NR(LEVEL) on all adjacent Up links
    Unmark all adjacent links except link L which is marked
-else if NUMBER = LEVEL mark link L
```

When a node executes A) above, we say that it ORIGINATES a restart; when it changes LEVEL and sends NR's in A) or B) we say that it RESTARTS. Before continuing with the description of the algorithm we make three definitions:

1) Two nodes X and Y are "joined" at some time if link (X,Y) is Up at both X and Y and there is no NR in transit on the link.

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 and if no NR is in transit on a link

outgoing from the set. A node is inactive if it belongs to an inactive set, else it is active.

We now complete the description of the algorithm, introducing the key element that prevents a monotonic increase in LEVEL; contrary to the algorithms mentioned in section 2 it does not rely at all on sending and tracking differences between restart numbers.

C) Whenever a node becomes inactive, it can arbitrarily change the value of LEVEL (e.g. reset it to 0).

To prove that the algorithm works correctly we characterize the set of its legal states. The state of the algorithm at any time includes the state of the nodes (Active or Inactive, value of LEVEL), the state of the links at a node (Down, unmarked, marked) and the set of messages in transit on the links.

Initially a node is inactive and isolated. Just after a link comes Up at a node, it is unmarked there, the node is active and a NR is in transit on the link.
The legal states for a link that is Up at both ends, X and Y, are listed in Table 1.
When link (X,Y) goes Down at X but is still Up at Y, the state of Y and the messages in transit to Y are those characterized by columns 2 and 4 in table 1.

To establish the correctness of this characterization of the legal states it is enough to notice that it is true initially, and remains true no matter what events occur.
We point out two properties that are important in proving the correctness:
Property 1: An inactive node can only become active by restarting, and not by having another node in its resynch set restart, as a node that restarts leaves its previous resynch set and becomes the single element in a new set. On the other hand a number of events (NR arriving at its destination, restart, link going Down) at remote nodes can cause an active node to become inactive.

Property 2: If a node is active, it can only restart at a higher level. This is not directly imposed by the description of the algorithm, which allows a restart at any level when a NR is received on a marked link. It holds as long as the state of the algorithm is legal (according to table I), as only NR's at a higher level can be received on a marked link at an active node.

Relying on those two properties it is easy but tedious to prove the validity of table I; the possible events in each legal state and the possible following states are listed in Table II.

We can now state the key theorem:

Theorem I.

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 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.
- From Table I if it has a neighbor Y at a lower level there is a NR in transit (with the highest level) to Y, which will eventually become active at the highest level; in the meantime X cannot become inactive.
- If Y is at the same level as X then either both are in the same resynch set, or there is a NR in transit from one of them on a link unmarked at the other. In any case one cannot become inactive without the other.

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. Also no such active node can restart, as it would be at a higher level (by Property 2); thus all NR's must stop flowing. Again from table 1 this implies that all nodes will be active with all their links marked, which leads to inactivity.

The previous theorem is important because of the following corollary that allows us to combine another distributed algorithm with the numbering algorithm just presented:

Corollary I

Assume a distributed algorithm that halts when executed in a network with fixed topology is started at each node of the network (with some initial conditions) each time a restart (in the numbering algorithm) occurs there, and assume that its messages are processed at another node only when they arrive on a marked link.
Then after a finite number of topological changes and spontaneous restarts the algorithm will halt at all nodes in the network in the same state as if it had run once on a network with the final topology (starting with the same initial conditions).

Proof:
This is clear from the previous theorem if the algorithm is started after all nodes become inactive. From the point of view of the algorithm messages it makes no difference if the algorithm is started instead at the beginning of the last restart, as no message flows from X to Y between the moment X restarts and the moment X becomes inactive.

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 it, but we start by introducing new concepts that will be useful later.

Assume that the nth time a node I restarts it becomes a "virtual node" with "virtual identity" (I,n) in a "virtual network". A directed "virtual link" appears from a virtual node (I,n) to a virtual node (J,m) when (if ever) one of the NR's sent 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".

Note the following:
- From table I if there is a virtual link from (I,n) to (J,m) and a virtual link from (J,m) to (I,o), then n = o.
- If I and J were joined during the nth restart at I and the mth restart at J, then there are virtual links in both directions between (I,n) and (J,m). The converse need not hold; in fact a virtual link may never have been up at both ends simultaneously.
- It is possible to have many virtual nodes corresponding to the same node in the same component of a "virtual network", but only one can be alive at any time.

4 A MINIMUM HOP SHORTEST PATH ALGORITHM.

The following distributed algorithm is based on ideas from [Gal76]. It can be seen as a partially synchronized implementation of the Ford-Bellman algorithm used in the original ARPANET routing procedure [McQ77]. When run on a network with fixed topology it stops with each node knowing what other nodes are in the same connected component at distance k (in hops), for all k. When run (as specified in Corollary 1) in conjunction with the numbering algorithm presented in section 3 it halts at a node ONLY when the node is inactive. It relies heavily on the assumption that all nodes have distinct identities. We first give a narrative outline, next follow it with a precise description and finally prove the main property.

Each node I maintains a vector D(I). Its Jth entry D(I)(J) is set to the minimum distance (in hops) from I to J; we will see that the distances will be measured in a virtual network and we should accept the possibility that many nodes with the same identity may be present in a connected component of that network.

Nodes exchange messages consisting of node identities. An identity J is included in the kth message from I to its neighbors if there is a node with identity J at distance k from I, and none closer. Link counters C(I)(K) serve to remember how many messages have been received on a link K at I. When all neighbors

of I have informed it of the identities of the nodes at distance k-1 from them, node I sets a counter HOP(I) to k, it finds the set T of all nodes at distance k, and it informs its neighbors by sending T.
If at some point T is empty and if nodes have distinct identities then all nodes in the connected network component have been discovered and the radius of the network (as seen by I) is HOP(I) - 1. However we cannot assume that all nodes have distinct identity (Figure 3). When T is empty, node I merely sets the variable R(I) to HOP(I) - 1 (i.e. what it assumes the radius to be) and keeps running the algorithm until HOP(I) is greater than three times R(I). If this ever happens then the node must be inactive! The proof that follows the formal description will make clear why this is so; we distinguish between a node stopping, i.e. setting its STATE to STOPPED, and the algorithm terminating, e.g. because no more messages are in transit.

```
     Shortest Path Algorithm at node I:

     A) Initially (whenever a restart occurs):
D(I)(I) = 0, D(I)(J) = 00, V J # I
C(I)(K)=0 V Up links K at I
HOP(I) = R(I) = 0
If no link is Up, STATE(I) = STOPPED else STATE(I) = WORKING

     B) When all Up adjacent links become marked:
Send {I} on all links

     C) Receive set S on link K while STATE(I) = WORKING
C(I)(K) = C(I)(K) + 1
for all J in S :  D(I)(J) = min (D(I)(J), C(I)(J))
If C(I)(K) > HOP for all Up links K, then
     HOP(I) = HOP(I) + 1
     Send T = {all nodes J | D(I)(J) = HOP(I)} on all Up links
     If R(I) = 0 and T = empty then R(I) = HOP(I) - 1
     If HOP(I) > 3 R(I) then
          send {} {} on all Up links (i.e.  two empty sets)
          reset LEVEL (in the numbering algorithm)
```

STATE(I) = STOPPED

We will now prove Theorem 2:

1. The previous algorithm terminates correctly when run on a network with fixed topology and unique node identities.

2. If the algorithm stops at a node I, then

2.1 I is inactive.

2.2 there was a time when all nodes J with D(I)(J) < 00 formed a single network component with all links marked at both ends (this is the "resynch" property that [Fin79] attempted to obtain).

It can be shown easily by induction on HOP (see [Gal76] or [Seg83]) that the D(.)'s are correctly set when the algorithm runs on a network with fixed topology and that at any time the values of HOP(.) at neighboring nodes that have not stopped differ by at most 1.
It is also easy to see that nodes with the smallest R will stop, after sending three empty messages (T and two others). This in turn guarantees that their neighbors, whose R differs by at most 1, will also stop; continuing the argument one sees that all nodes stop (Figure 2).

To handle the case of changing topology, consider the operation of the shortest path algorithm at a node I during a time interval between executions of step A, i.e. the algorithm as it executes at a virtual node (I,n). In particular consider the virtual network of that node, 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 D(I)(.), which are set in answer to message receptions, reflect the connectivity of the nodes in the virtual network. Also the values of HOP(.) at adjacent WORKING nodes can differ by at most 1, and this extends to nodes at

distance d.

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

A) If there is a dead node (J,s) at distance R((I,n)) or less at time t, we claim that (I,n) can never stop. Assume to the contrary that it is the first to eventually stop under these conditions and consider the situation at time t (Figure 3.a, I=1, J=3).

We first show that no node X on a shortest path between (I,n) and (J,s) can have stopped yet:
- if R(X) has not been set, X cannot have stopped
- if R(X) has been set then either (J,s) or another dead node closer to X must be within R(X) of X and node X cannot have stopped, as (I,n) was assumed to be the first to stop under these conditions.

It follows that HOP((J,s)) is within R((I,n)) of HOP((I,n)), i.e. not greater than 2 R((I,n)); it will never change, insuring that HOP((I,n)) will never exceed 3 R((I,n)) and that (I,n) will never stop.

B) If all virtual nodes at distance R((I,n)) 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((I,n)) + 1 from node (I,n) but those virtual nodes (if any) have the same ID as a live node and are thus dead (Figure 3.b, I=1, J=3).

Consequently those live nodes constitute an inactive resynch set. If the algorithm later stops at (I,n), I must still be inactive (property 1 in section 3).

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

If there is a dead node (J,s) at distance R((I,n)) + 1 from (I,n), there must be another node (J,t) (with t > s) at distance not greater than R((I,n)) from (I,n). Consider the moment where a path of length not exceeding 2 R((I,n)) + 1 joined (J,t) and (J,s) (this must occur). Some intermediate node on such a path had not executed step B in the shortest path algorithm and thus its neighbor toward (J,s) still had HOP(.) = 0, and all the nodes on the path were still active (thus they had not stopped). We can conclude that HOP((J,s)) must have been less than 2 R((I,n)) and this would prevent HOP((I,n)) from exceeding 3 R((I,n)) and stopping.
Figure 4 illustrates a number of scenarios that help understand the workings of the algorithm.

Before closing we make three observations:

1) Our goal in presenting the previous algorithms was to keep the discussion simple, not to minimize the number of messages or the time to completion. In particular, assuming that all nodes start simultaneously, that each message transmission requires one time unit and that processing time is negligible, the algorithm takes about 3 times the network diameter to complete in a fixed topology.

There exist various methods to reduce this to 2 times the diameter, which is about twice the time required by the (incorrect) algorithms mentioned in the introduction. One such method is for a node to broadcast a STOP message to its neighbors before stopping. On reception of this message, a node that has not stopped yet forwards the STOP message and stops.

2) There exist other algorithms that detect inactivity, but all those known to us contain a phase similar to what the shortest path algorithm does.

3) The fact that LEVEL is reset from time to time does not imply that it is bounded. However boundedness is easy to insure [Fin79] by not allowing a link to come Up and not originating spontaneous restarts when LEVEL is above a threshold. LEVEL can only increase above the threshold due to link failures and this guarantees its boundedness.

REFERENCES

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

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.

TABLE I: Legal states when a link (X,Y) is up at both ends

In all cases messages in transit are in order of increasing levels and the last message if any has the current level of its source.

State	State	State	ordering	number of messages	
Label	at X (*)	at Y (*)	of levels	in transit on: X -> Y	Y -> X
1	I,m	I,m	any	0	0
2.a	A,u	I,m	any	≥ 1	0
2.b	I,m	A,u	any	0	≥ 1
3.a	A,u	A,u	$X > Y$	≥ 1	≥ 0
3.b	"	"	$X < Y$	≥ 0	≥ 1
3.c	"	"	$X = Y$	≥ 1	≥ 1
4.a	A,u	A,m	$X = Y$	0	≥ 1
5.a	"	"	$X > Y$	≥ 1 (**)	≥ 0
4.b	A,m	A,u	$X = Y$	≥ 1	0
5.b	"	"	$X < Y$	≥ 0	≥ 1 (**)
6	A,m	A,m	$X = Y$	0	0

(*) I = node inactive; A = node active; u = link unmarked; m = link marked
(**) Message(s) in transit have level(s) greater than the destination level.

TABLE II: Verification of Table I

Events that can trigger a change in state at X, Y, or on the link (X,Y) and the possible resulting states are given next to each state.

States x.b are not treated explicitly, they behave as the corresponding x.a states with the roles of X and Y exchanged.

"Restart" means the beginning of a restart for a reason other than the reception of a message over the link
"Reception" means the processing of a message arriving on the link.

1	Restart at X: 2.a	Restart at Y: 2.b
2.a	Restart at X: 2.a	Restart at Y: 3.a, 3.b, 3.c
		Reception at Y: 4.a, 5.a
3.a	Restart at X: 3.a	Restart at Y: 3.a, 3.b, 3.c
	Reception at X: 3.a	Reception at Y: 2.a, 3.a, 4.a, 5.a
3.c	Restart at X: 3.a	Restart at Y: 3.b
	Reception at X : 3.c, 4.b	Reception at Y: 3.c, 4.a
4.a	Restart at X: 5.a	Restart at Y: 3.b
	Reception at X: 1, 4.a, 6	
5.a	Restart at X: 5.a	Restart at Y: 3.a, 3.b, 3.c
	Reception at X: 5.a	Reception at Y: 4.a, 5.a
	If no message is in transit from Y an event in Y's resynch set can cause a transition to 2.a.	
6	Restart at X: 5.a	Restart at Y: 5.b
	An event in the common resynch set can cause a transition to 1	

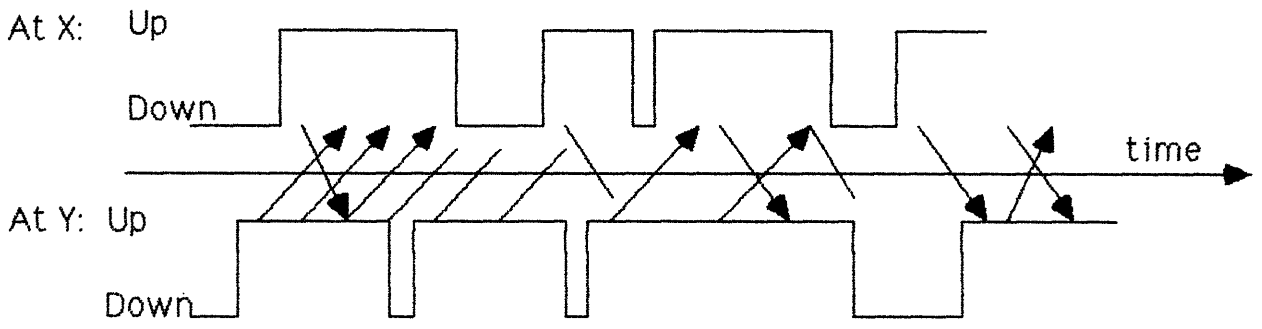
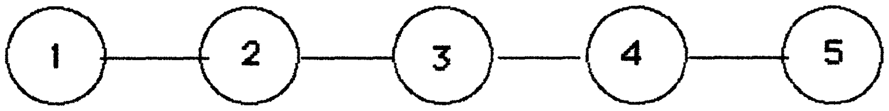


Figure 1

Illustration of link state history.
→ indicate successfull message transmissions.
—— indicate unsuccessfull transmissions.



HOP										
0	{1}		{2}		{3}		{4}		{5}	
1	{2}		{1,3}		{2,4}		{3,5}		{4}	
2	{3}		{4}		{1,5}		{2}		{3}	
3	{4}		{5}		{}	R=2	{1}		{2}	
4	{5}		{}	R=3	{}		{}	R=3	{1}	
5	{}	R=4	{}		{}		{}		{}	R=4
6	{}		{}		{}		{}		{}	
7	{}		{}		{}	Stop	{}		{}	
8	{}		{}		{}		{}		{}	
9	{}		{}		{}		{}		{}	
10	{}		{}	Stop			{}	Stop	{}	
11	{}		{}				{}		{}	
12	{}		{}				{}		{}	
13	{}	Stop							{}	Stop
14	{}								{}	
15	{}								{}	

Figure 2

The sets T sent by the nodes at different value of HOP are shown.
The instants where R is set are shown together with R's values
The instants where the nodes stop are also indicated, but the sending of the extra empty sets is shown at the subsequent values of HOP.

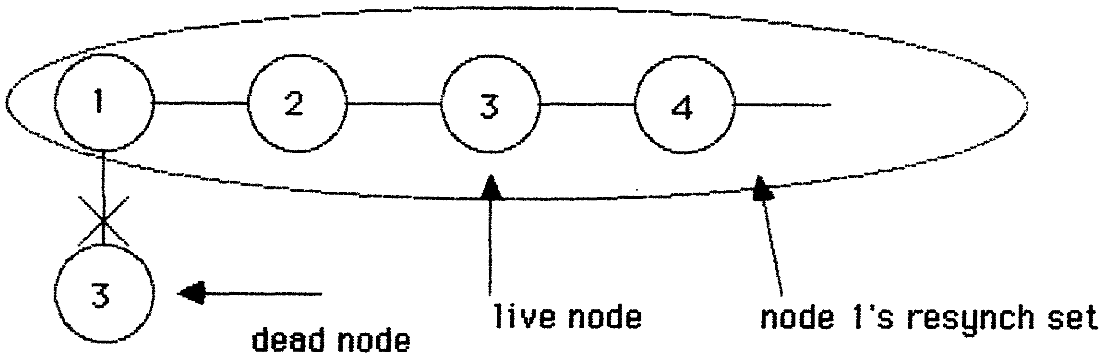


Figure 3 a

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.

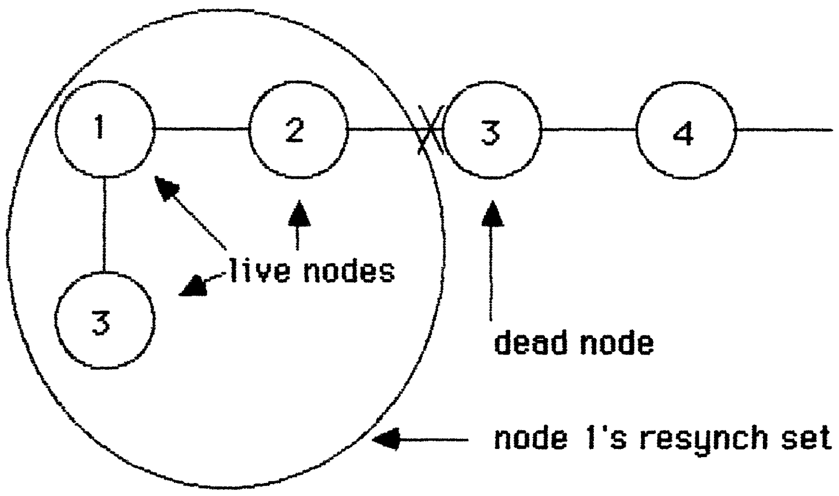
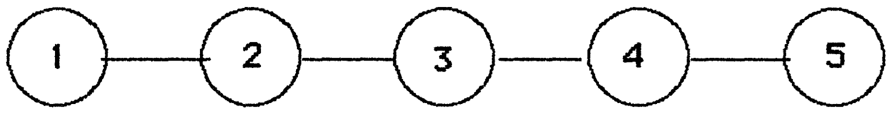


Figure 3 b

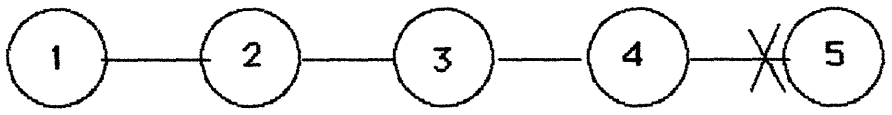
Node 3 appears twice in the virtual network,
R(1) is set to 1
nodes 1,2 and 3 may become inactive,
but they will not stop in the shortest path algorithm



HOP					
0		{2}	{3}	{4}	{5}
1			{4}	{3,5}	{4}
2				{2}	{3}
3					{2}

Figure 4

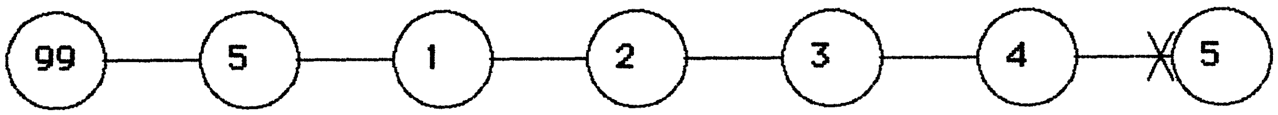
a) Initial phase
Messages initially exchanged.
Node 1 has not started the current update
After this initial phase link (4,5) fails
but node 4 is not notified



HOP					
0	{1}	{2}	{3}	{4}	{5}
1	{2}	{1,3}	{4}	{3,5}	{4}
2	{3}	{4}	{1,5}	{2}	{3}
3	{4}	{5}	{} R=2	{1}	{2}
4	{5}	{} R=3	{}	{} R=3	
5	{} R=4	{}	{}		
6	{}	{}			
7	{}				

b) Scenario 1:
Node 1 eventually starts the update.
No more topological change occurs

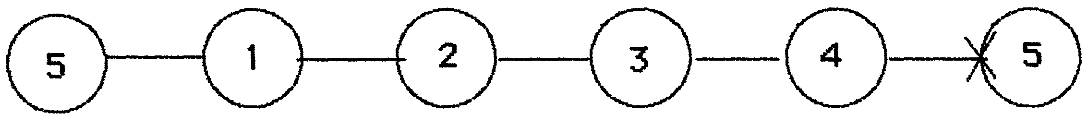
All the nodes were inactive when their R was set.
None of the node stops



HOP							
0	{99}	{5}	{1}	{2}	{3}	{4}	{5}
1	{5}	{1,99}	{2,5}	{1,3}	{2,4}	{3,5}	{4}
2	{1}	{2}	{3,99}	{4,5}	{1,5}	{2}	{3}
3	{2}	{3}	{4}	{99}	{} R=2	{1}	{2}
4	{3}	{4}	{} R=3	{} R=3	{99}	{} R=3	
5	{4}	{} R=4	{}	{}	{}		
6	{} R=5	{}	{}	{}			
7	{}	{}	{}				
8	{}	{}					
9	{}						

c) Scenario 2
Node 5 connects and completes an update with 99,
then connects with 1

Node 2 was not inactive when its R was set
None of the nodes stops



HOP						
0	{5}	{1}	{2}	{3}	{4}	{5}
1	{1}	{2,5}	{1,3}	{4}	{3,5}	{4}
2	{2}	{3}	{4,5}	{1,5}	{2}	{3}
3	{3}	{4}	{} R=2	{} R=2	{1}	{2}
4	{4}	{} R=3	{}	{}	{} R=3	
5	{} R=4	{}	{}	{}		
6	{}	{}	{}			
7	{}	{}				
8	{}					

d) Scenario 3
Node 5 connects to node 1.
Both start the update

All nodes are inactive when their R is set.
None of the nodes terminates, although
node 2 was close.